MaxonC++

Das Entwicklungssystem zur objektorientierten Programmierung

AMIGA

AMIGA

MaxonC++

Das C++ Entwicklungssystem

MAXON
computer

MaxonC++

Autoren:

C/C++ Compiler	**Jens Gelhar**
Edward	Thomas E. Wieger
Maxon Debugger	Christian Spreuer
MakeAPP/RCT.Library	Jürgen Haage
MaxonASM	Klaus-Dieter Sommer
HotHelp	Hartmut Stein

© 1992 by MAXON Computer

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung der Programme, der Anleitungen sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitungen enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

Endlich - **Maxon**C++ ist da!

Die Welt der professionellen Softwareentwicklung erlebt in diesen Jahren eine stille Revolution: Das Konzept der objektorientierten Programmierung setzt sich in allen Bereichen durch.

Nachdem Simula in den späten 60er Jahren ihrer Zeit zu weit voraus war und Smalltalk später zu sehr mit allen vertrauten Konzepten der Programmierung brach, um sich durchsetzen zu können, gibt es jetzt endlich eine objektorientierte Programmiersprache, die sich als der neue Standard in der Softwareentwicklung etabliert - die Rede ist natürlich von C++.

Es gibt viele Gründe, die gerade für C++ sprechen. Zunächst setzt diese Sprache auf C auf, der im professionellen Bereich bisher verbreitetsten Sprache. Das geht so weit, daß C-Quelltexte sich mit minimalen oder ganz ohne Modifikationen mit einem C++-Compiler übersetzen lassen. Diese Abwärtskompatiblität erleichtert natürlich den Umstieg von C nach C++ gewaltig und macht C++ für jeden C-Programmierer zur ersten Wahl.

Ferner wurde C++ konsequent so konzipiert, daß eine objektorientierte Lösung weder beim Speicherplatz noch in Bezug auf die Laufzeit einer konventionellen Lösung unterlegen ist und so die Effizienz, die man von C gewohnt ist, erhalten bleibt.

Nicht zuletzt ist zu erwähnen, daß C++ von den AT&T Bell Laboratories entwickelt wurde und von AT&T, also gewiß keiner kleinen Firma, gefördert wird. Schon allein dieser Umstand hat für die Verbreitung von C++ im UNIX-Bereich und bei den sich mehr und mehr am Workstation-Markt orientierenden Microcomputer gesorgt.

Nur in der Amiga-Welt war von all diesen aufregenden Entwicklungen lange Zeit nichts zu spüren. Während auf anderen Systemen sogar jeder bessere C-Compiler objektorientierte Konzepte von C++ übernahm oder gleich zu einem vollständigen C++-System ausgebaut wurde, empfand der Amiga-User schon das Erscheinen von ANSI-kompatiblen C-Compilern als Fortschritt. Da leistungsfähige Compiler aber die Voraussetzung für professionelle Programmentwicklung sind, litt unter diesem Mangel nicht nur der persönliche Komfort der Amiga-Programmierer, sondern auch die Entwicklung der Amiga-Software allgemein wurde behindert und drohte den Anschluß an konkurrierende Systeme endgültig zu verlieren. Es war höchste Zeit, daß etwas geschah.

Nach mehr als einjähriger Entwicklungszeit und einer umfangreichen Debugging-Phase ist Maxon C++ jetzt einsatzbereit. Maxon C++ ist keine Portierung von einem anderen System, sondern wurde speziell für den Amiga (und auf einem Amiga) entwickelt und auf die Besonderheiten dieses Systems zugeschnitten. Der Compiler ist zum „2.0"-Standard kompatibel und erzeugt aus dem C++-Quelltext direkt ausführbaren Code, ohne Umwege über C, Assembler oder andere Sprachen. Trotzdem läßt er sich auf ANSI-C-Emulation umschalten, um volle Kompatiblität zu den vorhandenen C-Compilern zu erreichen, und kann optional auch Assembler-Quelltext für die Weiterverarbeitung mit einem fremden Assemblern erzeugen.

Der Compiler wird in gleich zwei Versionen geliefert: Eine um einen Editor aufgebaute, integrierte Entwicklungsumgebung bietet höchsten Komfort, gleichermaßen für das bequeme Erstellen kleiner Programme wie für die Entwicklung umfangreicher Projekte. Eine konventionelle Kommandozeilen-Version erleichtert die Integration in existierende Umgebungen und Tools.

Lieferumfang

Das MaxonC++ Compilersystem besteht aus sieben Handbüchern, die durch Einlageblätter voneinander getrennt sind:

1. MaxonC++ Vorwort
2. MaxonC++ Anwender-Handbuch
3. Edward
4. MaxonC++ Tutorial
5. MaxonC++ Referenz-Handbuch
6. Maxon MakeApp
7. Maxon HotHelp

Die Developer-Version des Compilersystems ist um zwei Handbücher erweitert:

8. Maxon Source-Level-Debugger
9. MaxonASM (CLI-Version)

Der Compiler und die Zusatzprogramme werden auf mehreren Disketten geliefert. Genaue Angaben zur Installation und zur Diskettenanzahl entnehmen Sie bitte der „Readme"-Datei auf der Installationsdiskette. Da sich nach der Drucklegung noch einige Änderungen, was die Diskettenanzahl und die Programmzusammenstellung angehen, ergeben können, ist dies der beste Weg, um das Handbuch so aktuell wie möglich zu halten.

Developer-Upgrade

Allen registrierten Besitzern der normalen MaxonC++ Compiler-Version bieten wir die Möglichkeit eines Upgrades auf die MaxonC++-Developer-Version. Das Upgrade beinhaltet zwei weitere Disketten und die beiden oben beschriebenen Zusatzhandbücher. Die Upgradekonditionen erfragen Sie bitte bei Maxon Computer.

Registrierung

Ihrem Compilersystem liegt eine Registrierkarte bei, die Sie umgehend an uns zurücksenden sollten. Sie gewährleisten dadurch, daß Sie regelmäßig über Produktneuerungen und Neuerscheinungen informiert werden. Desweiteren steht Ihnen dann unsere tägliche Hotline für Problemlösungen zur Verfügung.

Hotline und Support

Unsere Hotline ist werktags zwischen 15:00 und 17:00 Uhr unter der Rufnummer 06196 / 43780 zu erreichen. Bitte prüfen Sie vor einem Anruf genau den Problemfall:

- Haben Sie die Software richtig installiert?
- Benutzen Sie die richtige Workbench Version (nicht unter 1.3)?
- Laufen irgendwelche Programme im Hintergrund (Screenblanker, Maustools o.ä.)?

Bitte haben Sie Verständnis, daß wir keine Garantie für das Zusammenspiel mit anderer Software geben können. Besonders einige Tools, die in der Startup-Sequence mitgestartet werden, können die Arbeitsfähigkeit sehr stören.

Bei einem Anruf halten Sie bitte Ihre Registriernummer, die Versionsnummer des jeweiligen Softwarepaketes und eine genaue Konfigurationsbeschreibung bereit. Die Registriernummer ist auf den Orginal-Disketten aufgedruckt. Die Versionsnummer finden Sie entwender im Info-Dialogfenster des jeweiligen Programmes oder wenn Sie (ab Amiga OS 2.0) den Befehl:

```
Version <Programmname> full
```

in der Shell (CLI) eingeben.

Wenn Sie uns schreiben, benötigen wir zum zügigen Bearbeiten, ebenfalls die beschriebenen Angaben.

Richten Sie Ihr Anschreiben an folgende Adresse:

Maxon Computer GmbH
"MaxonC++-Support"
Schwalbacher Straße 52
W-6236 Eschborn

Literaturangaben

Wenn Sie bereits in einer anderen Programmiersprache als C++ programmieren können, ist das Tutorial-Handbuch eine sehr gute Einführung für Sie, um die Syntax und den Umgang mit C++ kennenzulernen. Als Programmieranfänger empfiehlt es sich jedoch, ein Grundeinführungsbuch durchzuarbeiten. Die nachfolgende Liste enthält eine Auswahl an Büchern zu den Themen objektorientierte Programmierung im allgemeinen, Programmierung in C und C++ und zur Programmierung des Amiga Betriebssystems.

Objektorientierte Programmierung allgemein

Objektorientierte Programmierung - Eine Einführung
R. Engel - 338 Seiten - ISBN 3-89090-353-3
Markt&Technik Verlag AG/Haar

C und C++ Literatur

Lernen und Üben: C-Schulung
H. Niegemann und T. Schäfer - 354 Seiten - ISBN 3-89090-218-9
Markt&Technik Verlag AG/Haar

Die Einführung: C
S. Baloui - 379 Seiten - ISBN 3-89090-334-7
Markt&Technik Verlag AG/Haar

Programmieren in C++
E. Huckert - 348 Seiten - ISBN 3-89090-171-9
Markt&Technik Verlag AG/Haar

Clever Programmieren: C++
M. Haun - 352 Seiten - ISBN 3-88322-327-1
IWT Verlag GmbH/Vaterstetten

IWT Kompakt: C-Referenz für alle ANSI-C-Compiler
P. Prinz - 304 Seiten - ISBN 3-88322-340-9
IWT Verlag GmbH/Vaterstetten

C++ lernen und beherrschen: Einführung und Leitfaden
S. B. Lippmann - ca. 500 Seiten - ISBN 3-89319-375-8
Addison-Wesley (Deutschland) GmbH/Bonn

Die C++ Programmiersprache
B. Stroustrup - 380 Seiten - ISBN 3-89319-386-3
Addison-Wesley (Deutschland) GmbH/Bonn

The C++ Programming Language
B. Stroustrup - 576 pages - ISBN 0-201-53992-6
Addison-Wesley (Deutschland) GmbH/Bonn

The Annotated C++ Reference Manual
M. A. Ellis/ B. Stroustrup - 400 pages - ISBN 0-201-51459-1
Addison-Wesley (Deutschland) GmbH/Bonn

Die Kunst der objektorientierten Programmierung - Prinzipien und Konzepte in C++
A. Schäpers - ca. 440 Seiten - ISBN 3-89319-226-3
Addison-Wesley (Deutschland) GmbH/Bonn

C-Programmieren: Ein Kurs zum Selbststudium
G. Krüger - ca. 500 Seiten - ISBN 3-89319-372-3
Addison-Wesley (Deutschland) GmbH/Bonn

Der C-Experte: Programmieren ohne Pannen
A. Koenig - 190 Seiten - ISBN 3-89319-233-6
Addison-Wesley (Deutschland) GmbH/Bonn

Debugging C
R. Ward - 322 Seiten - ISBN 3-925118-90-X
Addison-Wesley (Deutschland) GmbH/Bonn

C Professional Edition
W. Herglotz - ca 450 Seiten - ISBN 3-89360-004-3
BHV Verlags GmbH/4052 Korschenbroich

Das Einsteigerseminar C
W. Herglotz - 224 Seiten - ISBN 3-89360-606-8
BHV Verlags GmbH/4052 Korschenbroich

Das Einsteigerseminar C++
W. Herglotz - 224 Seiten - ISBN 3-89360-622-X
BHV Verlags GmbH/4052 Korschenbroich

C++ für Programmierer
F. Bause/ W. Tölle - 286 Seiten - ISBN 3-528-05110-8
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/6200 Wiesbaden

Einführung in die Programmiersprache C++
F. Bause/ W. Tölle - 171 Seiten - ISBN 3-528-14689-3
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/6200 Wiesbaden

Referenzhandbuch Standart C
P.J. Plauger/J. Brodie - 236 Seiten - ISBN 3-528-04741-0
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH/6200 Wiesbaden

C++ Einführung in die objektorientierte Programmierung
S. Prata - 750 Seiten - ISBN 3-89362-701-4
tewi Verlag

Der schnelle Umstieg auf C++
M. Aupperle - 76 Seiten (Chip Spezial) - ISBN 3-8023-1088-8
Vogel Verlag und Druck KG/8700 Würzburg

C für Umsteiger
B. Klöppel - 122 Seiten (Chip Spezial) - ISBN 3-8023-1081-0
Vogel Verlag und Druck KG/8700 Würzburg

Amiga Bücher

Amiga-Programmierrichtlinien
H. Gzella - 272 Seiten - ISBN 3-87791-049-1
Markt&Technik Verlag AG/Haar

Amiga ROM Kernel Reference Manual: Devices
512 pages - ISBN 0-201-56775-X
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Libraries
960 pages - ISBN 0-201-56774-1
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga ROM Kernel Reference Manual: Includes and Autodocs
1000 pages - ISBN 0-201-56773-3
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga Hardware Reference Manual
512 pages - ISBN 0-201-56776-8
Addison-Wesley (Deutschland) GmbH/Bonn

Amiga User Interface Style Guide
224 pages - ISBN 0-201-57757- 7
Addison-Wesley (Deutschland) GmbH/Bonn

Maxon C++

Anwender-Handbuch

MAXON computer

Inhaltsverzeichnis

1. Ein erster Überblick 7

1.1 Die Komponenten des MaxonC++ Systems 7

1.2 Installation 7

1.3 Erste Schritte mit „MaxonC++“ 7

2. Die integrierte Entwicklungsumgebung 9

2.1 Programmstart und Optionen 9

2.2 Compiler und Editor - eine harmonische Beziehungskiste 10

2.3 Das Konzept 12

2.4 Compiler-Funktionen 13
2.4.1 Programme übersetzen 13
2.4.1.1 Das Compiler-Fenster 13
2.4.1.2 Abbruch und Ausstieg 15
2.4.1.3 Auswahl von Fehlermeldungen 16
2.4.1.4 Vom Compiler in den Editor 16
2.4.1.5 „Continue“ und „Run“ 17
2.4.2 Ausführen und Abbrechen 17
2.4.3 übersetzen von Dateien 18
2.4.4 Exe- und Objektdateien 18
2.4.5 Daten und Informationen 19

2.5 Features und Optionen 21
2.5.1 Ein kurzer Überblick 21
2.5.2 Compiler-Optionen 21
2.5.2.1 Include-Dateien 21
2.5.2.2 Der Arbeitsspeicher des Compilers 22
2.5.2.3 Vordefinieren von Symbolen 23
2.5.2.4 Diverse kleine, aber trotzdem wichtige Schalter 23
2.5.2.4.1 C++ 23
2.5.2.4.2 Unterbrechen 24
2.5.2.4.3 IntMult 32 24
2.5.2.4.4 Debug-Datei 24

2.5.2.4.5 Symbol-Hunk 24
2.5.2.4.6 Asm schreiben 25
2.5.2.4.7 Source in Asm 26
2.5.2.4.8 Optimieren 26
2.5.2.5 Optionale Warnungen und Meldungen 26
2.5.3 Das Linker-Dialogfenster 28
2.5.3.1 Datenmodelle 29
2.5.3.2 Linken oder nicht Linken - das ist hier die Frage 30
2.5.3.3 Bibliotheken 31
2.5.3.4 Zusammenfassen von Hunks 31
2.4.3.5 Schreiben von Dateien 32
2.5.4 „MaxonC++"-Systemeinstellungen 32
2.5.4.1 Autosave 32
2.5.4.2 Erzeugen von Icons 33
2.5.4.3 „MaxonC++" - ein polyglottes System 33
2.5.4.4 Protokollieren von Ein- und Ausgaben 33
2.5.4.5 Aufruf eines externen Assemblers 34
2.5.4.6 Argumente aus der imaginären Kommandozeile 34
2.5.4.7 Compiler- und Laufzeit-Fenster 34
2.5.4.8 Stackgröße 35
2.5.5 Aktuelles Verzeichnis 35
2.5.6 Voreinstellungen mit der „Config"-Datei 35

3. MAKE - eine integrierte Projektverwaltung 39

3.1 Das Prinzip 39
3.1.1 Wozu eine Projektverwaltung? 39
3.1.2 Was ist das überhaupt - ein Projekt? 39
3.1.3 Was hat das ganze mit „Make" zu tun? 40

3.2 Die Praxis 41
3.2.1 Erstellen von Projekten 41
3.2.2 Hinzufügen von Dateien, umständliche Methode 41
3.2.3 Weitere Funktionen 43
3.2.4 Externe Übersetzungen 44

3.3 Handgemachte Makedateien 45

4. Der Linker 48

4.1 Prinzipielles 48

4.2 Standard-Funktionen und das „Logfile" 50

4.3 Das Hauptprogramm und seine Argumente 52

4.4 Linker-Bibliotheken 53

4.5 Initialisierungen und das große Aufräumen 54

4.6 Mit und ohne Startup-Code 54

5. Die CLI-Version „CPPC“ 57

5.1 Das Konzept 57

5.2 Optionen 58
5.2.1 Allgemeines 58
5.2.2 Ergebnisse und Ausgaben 59
5.2.2.1 Assembler-Quelltext 59
5.2.2.2 Objektdateien: -c 59
5.2.2.3 Ausführbare Programmdatei: -o 60
5.2.3 Preprozessor 60
5.2.3.1 Definieren von Symbolen: -d 60
5.2.3.2 Includedateien: -i 61
5.2.4 Compiler-Modus: -pc und -pp 61
5.2.5 Codeerzeugung 62
5.2.5.1 Breakpoints setzen: -gb 62
5.2.5.2 Datenmodell: -gd und -gD 62
5.2.5.3 Zusammenfassen von Hunks: -gh 62
5.2.5.4 Ganzzahlige Arithmetik: -gm 62
5.2.5.5 Optimierungen: -go und -gO 63
5.2.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw 63
5.2.6 Debugger 63
5.2.6.1 Maxon Debugger: -b und -bi 63
5.2.6.2 Symbolischer Debugger: -bs 63
5.2.7 Warnungen und Fehler 63
5.2.7.1 Alternative Fehlerbehandlung: -e und -en 63
5.2.7.2 Optionale Warnungen: -w 64
5.2.8 Linker 65
5.2.8.1 Library-Pfade: -j 65
5.2.8.2 Linker-Libraries: -l 65
5.2.9 Die integrierte Projektverwaltung: -m 65
5.2.10 Arbeitsspeicher: -w 66

5.3 Voreinstellungen mit „CPPOPT“ 66

5.4 Zusammenfassung 67

1. Ein erster Überblick

1.1 Die Komponenten des MaxonC++ Systems

MaxonC++ wird auf mehreren Disketten ohne Kopierschutz geliefert. Die genaue Anzahl und den Inhalt der Disketten erfahren Sie, wenn Sie die „Readme"-Datei auf der ersten Diskette lesen. Da sich nach der Drucklegung dieses Handbuches noch Änderungen ergeben können, wählten wir diesen Weg.

1.2 Installation

Um das System auf Ihrer Festplatte zu installieren, sollten Sie zunächst ein geeignetes Verzeichnis anlegen, beispielsweise mit dem Namen „dh0:MCPP". Sie sollten Datei- oder Verzeichnisnamen mit einem „+" am Ende meiden und das Verzeichnis nicht unbedingt „C++" nennen. Starten Sie jetzt das Installationsprogramm von der ersten Diskette. Alles weitere erfahren Sie aus der „Readme"-Datei.

1.3 Erste Schritte mit „MaxonC++"

Die integrierte Entwicklungsumgebung „MCPP" kann wahlweise von der Workbench oder vom CLI gestartet werden. Dabei besitzt das Programm keine eigene Benutzeroberfläche, sondern lädt den Editor „Edward" nach. Folglich muß dieser Editor korrekt installiert sein. Wird der Edward nicht gefunden, wird „MaxonC++" gleich mit einem entsprechenden Hinweis beendet.

Nach dem Start landen Sie also automatisch im Editor. Hier können Sie Ihre C++-Programme schreiben und bearbeiten. Dem „Edward" ist ein separates Handbuch gewidmet, deshalb soll hier nicht näher auf die Editor-Funktionen eingegangen werden.

Am besten machen Sie sich jetzt ein wenig mit dem Editor vertraut, indem Sie ein kleines Programm schreiben. Sie könnten z. B. folgendes unvermeidliche und traditionsreiche Prgrämmchen abtippen:

```
#include <stream.h>

void main()
{
  cout « "Hello, World!\n";
}
```

Nun haben Sie ein Programm als Quelltext vorliegen und können den Compiler ausprobieren. Zuerst sollten Sie den Text abspeichern, um ihm so einen sinnvollen Namen zu geben, mit dem der Compiler auch etwas anfangen kann. Dann wählen Sie einfach den Menüpunkt „Übersetzen" aus der Menüleiste und warten ab, was geschieht.

Nun öffnet sich ein Fenster, in dem „MaxonC++" einige mehr oder weniger kryptische Meldungen ausgibt. Tritt während der Übersetzung ein Fehler auf, wird er in diesem Fenster gemeldet und der entsprechende Quelltextbereich angezeigt. Sie können dann eine Fehlermeldung anwählen und mit den Symbolen „Edit" und „Close&Edit" direkt die entsprechende Stelle im Editor anspringen.

Wenn das System korrekt installiert wurde und Ihr Programm ebenfalls fehlerlos ist, sollte das Fenster aber ebenso schnell verschwinden, wie es aufgetaucht ist. Nun können Sie die Menü-Funktion „Ausführen" anwählen und das Programm starten. Auch dabei öffnet sich ein Fenster, in dem (wer hätte das gedacht!) „Hello, World!" ausgegeben wird. Außerdem werden Sie freundlich aufgefordert, die „Return"-Taste zu drücken, denn sinnvollerweise schließt „MaxonC++" das Fenster nicht, bevor Sie Gelegenheit hatten, die Ausgabe zu lesen.

Sie können das Programm natürlich noch einmal starten. Wenn Sie aber seit der letzten Übersetzung etwas am Quelltext verändert haben, wird automatisch der Compiler erneut gestartet.

2. Die integrierte Entwicklungsumgebung

2.1 Programmstart und Optionen

Wenn Sie „MCPP“ vom CLI bzw. aus einer Shell starten, können Sie optional einige Parameter angeben. Als erstes können Sie einen Dateinamen angeben, z. B.

```
run MCPP C++:prg/hello.c
```

Sofern die angegebene Datei existiert und lesbar ist, wird sie in den Editor geladen. Da der Edward mehrere Texte verwalten kann, können Sie natürlich auch mehrere Dateinamen angeben.

Wenn Sie eine andere Konfigurationsdatei als „S:MaxonC++.config“ benutzen wollen, können Sie ein zusätzliches Argument mit vorangestelltem „-c“ angeben. Ein Beispiel:

```
run MCPP test.c -cC++:spezial.config
```

Alternativ können Sie auch „c=XXX“ oder „-c=XXX“ setzen. Wichtig ist aber, daß hinter dem „c“ kein Leerraum steht, denn der Parameter „-c“ allein veranlaßt das Programm, keine Konfigurationsdatei zu laden und die jeweiligen Defaultwerte zu verwenden.

MaxonC++ besitzt eine integrierte Projektverwaltung, allgemein bekannt als „Make“. Sie können gleich beim Programmstart eine solche Make-Datei öffnen, und zwar mit der Option „-m“:

```
run MCPP -m plot.makefile
```

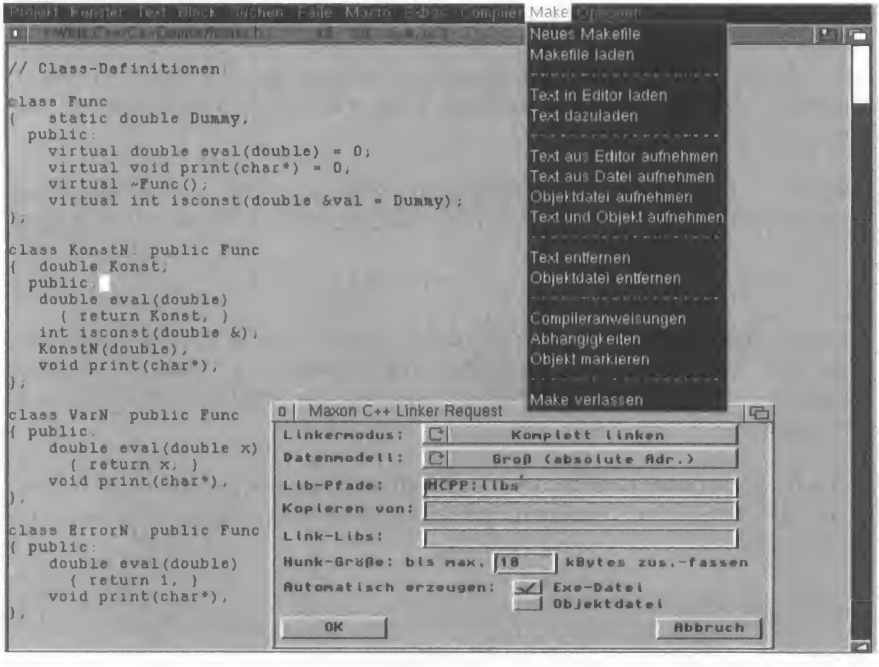
Wie Sie sehen, darf bei dieser Option Leerraum zwischen „-m“ und dem Dateinamen stehen. Auch hier sind wieder die Varianten „m=“ und „-m=“ erlaubt.

Natürlich lassen sich diese Optionen und Argumente in beliebiger Reihenfolge kombinieren, z. B.

```
"MCPP" fclass.c -m plot.makefile -c=df1:s/MaxonC++.config
```

2.2 Compiler und Editor - eine harmonische Beziehungskiste

Wie oben schon angedeutet, ist der Editor „Edward" ein eigenständiges Programm, das auch ohne den Compiler benutzt werden kann. „MaxonC++" lädt ihn einfach nach und startet ihn als zweiten Prozess, läßt aber keinen Zweifel daran aufkommen, daß er der Boß ist.



Weniger blumig ausgedrückt, richtet „MaxonC++" einen Port ein, über den sich Editor und Compiler Nachrichten zuschicken. Wird im Edward vom Benutzer eine Compiler-Funktion aufgerufen, etwa durch eines der Pull-Down-Menüs oder eine entsprechend belegte Tastenkombination, läßt Edward dem Compiler eine entsprechende Nachricht zukommen.

Ich quäle Sie mit diesen implementatorischen Details, damit Sie verstehen können, wie Sie das „MaxonC++"-System selbst konfigurieren können. Der Schlüssel dazu ist die „.DEF"-Datei des Edward. Hier werden alle Menüs und Tastenbelegungen des Editors abgelegt, und hier muß irgendwie die Benutzerschnittstelle des Compilers integriert werden, auch wenn der Editor nicht die geringste Anhnug hat, was ein Compiler ist und wozu er gut ist.

Deshalb hat der Edward eine Spezial-Funktion namens „call", die als Argument wiederum eine Compiler-Funktion erwartet. Beispielsweise gibt es eine „MaxonC++"-Anweisung namens „compile". Wird nun vom Benutzer, also Ihnen, im Edward ein Menü ausgewählt, das mit der Funktion „call compile" belegt ist, schickt Edward das „compile" als Nachricht an den „MaxonC++" und kümmert sich dann nicht weiter drum. „MaxonC++" dagegen erhält diese Anweisung, fordert vom Edward noch ein paar zusätzliche Daten an und beginnt dann freudig mit der Übersetzung. Natürlich kann man nicht nur ein Menü, sondern auch eine Taste oder Tastenkombination mit diesen „call"-Anweisungen belegen.

Auf diese Weise sind der Compiler und der Editor vollkommen voneinander abgekoppelt, und Edward braucht nichts über „MaxonC++" zu wissen. Nun kommt es natürlich vor, daß Sie den Edward einmal „einfach so" ohne Compiler benutzen wollen. Deshalb läßt Edward einen kleinen „Preprozessor" auf das „.DEF"-File los und filtert je nach Modus Teile aus dem Text aus - ähnlich wie ein C-Preprozessor bei „#ifdef"-Zeilen. Auch dieses Feature ist in Ihrem Edward-Handbuch beschrieben. „MaxonC++" teilt dem Edward mit, daß ein Symbol namens „COMPILER" definiert sein soll. Nun können Sie Dinge wie „#ifdef COMPILER" in Ihr Defs-File schreiben, um Definitionen je nach Modus ein- oder auszublenden. Insbesondere werden Sie sich um sämtliche Menü-Definitionen des Compilers eine „#ifdef COMPILER ... #endif"-Klammer legen wollen, wie es auch im mitgelieferten Standard-File der Fall ist.

Langer Rede, schwacher Sinn: Wenn ich im folgenden von irgendwelchen Menüs schwafele, unterschlage ich dabei jedesmal Hinweise wie „...oder wie auch immer Sie diese Funktion konfiguriert haben". Ich meine damit jeweils immer die Funktionen, die in der Standard-Konfiguration auf dem entsprechenden Menü liegen.

Es gibt übrigens standardmäßig folgende Menüs, mit den nachstehend in voller Länge und der jeweiligen Kurzform angegebenen Compiler-Funktionsnamen:

- Compiler
  - Übersetzen — compile — c
  - Ausführen — run — r
  - Abbrechen — break — b
  - Compile Datei — compfile — f
  - Exe-Datei — exefile — x
  - Objektdatei — objectfile — o
  - Information — info — i

- Make
  - Neues Makefile — newmk — nm
  - Makefile laden — loadmk — lm
  - Text in Editor laden — opentext — ot
  - Text aus Editor aufnehmen — addedittx — ae

- Text aus Datei aufnehmen	addfiletx	af
- Objektdatei aufnehmen	addobj	ao
- Text und Objekt aufnehmen	addtxobj	ato
- Text entfernen	remtx	rt
- Objektdatei entfernen	remobj	ro
- Compileranweisungen	mkinstr	in
- Abhängigkeiten	mkdep	dp
- Objekt markieren	touch	to
- Make verlassen	leavemk	qm

- Optionen

- System	opsys	os
- Compiler	opcomp	oc
- Linker	oplink	ol
- Verzeichnis	chdir	d
- Optionen sichern	saveconf	as

2.3 Das Konzept

Auf die Gefahr hin, daß Sie mich dafür steinigen, werde ich trotzdem noch ein paar grundsätzliche Worte loslassen. Prinzipiell kann „MaxonC++" in zwei verschiedenen Modi arbeiten: Im „normalen" Modus, in dem man vor allen kleinere Programme (mit nur einem Modul) schreibt, und im „Make"-Modus. Da der Projekverwaltung „Make" ein eigenes Kapitel gewidmet ist, befassen wir uns zunächst nur mit dem Standard-Modus, der natürlich auch aktiv ist, wenn Sie das „MaxonC++"-System starten.

Auch wenn Edward beliebig viele Texte gleichzeitig bearbeiten kann, verwaltet „MaxonC++" immer nur ein Programm. Das hat zum einen Gründe, die mit dem Speicherplatz zusammenhängen, und resultiert zum anderen aus der Beobachtung, daß das in der Praxis nicht besonders relevant ist, weil man normalerweise zwar durchaus mit mehreren Texten, aber an nur einem Programm arbeitet.

Die Funktion „compile" übersetzt stets den Text im gerade aktiven Fenster. Bei „run" wird immer zuerst folgendes geprüft:

- Liegt überhaupt ein übersetztes Programm vor?
- Befindet sich der Quelltext dieses Programms im aktiven Editor-Fenster?
- Blieb er seit der letzten Übersetzung unverändert?

Falls eine der drei Fragen mit „Nein" beantwortet werden muß, wird die „compile"-Funktion aufgerufen und damit der Text im aktiven Fenster übersetzt.

So, das sollte gesagt werden, damit Sie beim Rumprobieren mit „MaxonC++" (was Sie wahrscheinlich schon längst getan haben) nicht in vermeidbare Verwirrung stürzen (was dabei hoffentlich nicht geschehen ist). Und nun folgt endlich eine ausführliche Beschreibung der Funktionen von „MaxonC++":

2.4 Compiler-Funktionen

2.4.1 Programme übersetzen

2.4.1.1 Das Compiler-Fenster

Am besten fangen wir mit der Funktion an, wegen der Sie „MaxonC++" überhaupt gekauft haben, nämlich dem Übersetzen von Programmen.

Wenn Sie diese Funktion auswählen, wird - je nach Modus - entweder das Programm im gerade aktiven Editor-Fenster übersetzt, oder das integrierte „Make" entscheidet selbst, welche Objektdateien neu übersetzt werden sollen. In jedem Fall öffnet sich aber ein Fenster, das in fünf Bereiche aufgeteilt ist.

Am oberen Rand befindet sich die Statuszeile. Hier wird stets links der Name der gerade überstzten Datei angezeigt, während „MaxonC++" rechts sagt, was gerade getan wird bzw. zuletzt getan wurde.

Am besten probieren Sie es einmal aus: Laden Sie einen C-Quelltext (z. B. eines der Beispielprogramme) in den Editor und wählen Sie den Menüpunkt „Übersetzen" aus. Wenn alles klappt, wird oben rechts in schneller Folge (bei kurzen Programmen wohl zu schnell zum Mitlesen) das folgende Ausgegeben:

- „Compiling": Der Compiler liest und analysiert das Programm.

- „No Errors" bzw. „X Errors found": Nachdem der Quelltext gelesen und auf seine Korrektheit geprüft wurde, erzeugt der Compiler nun den Code dazu.

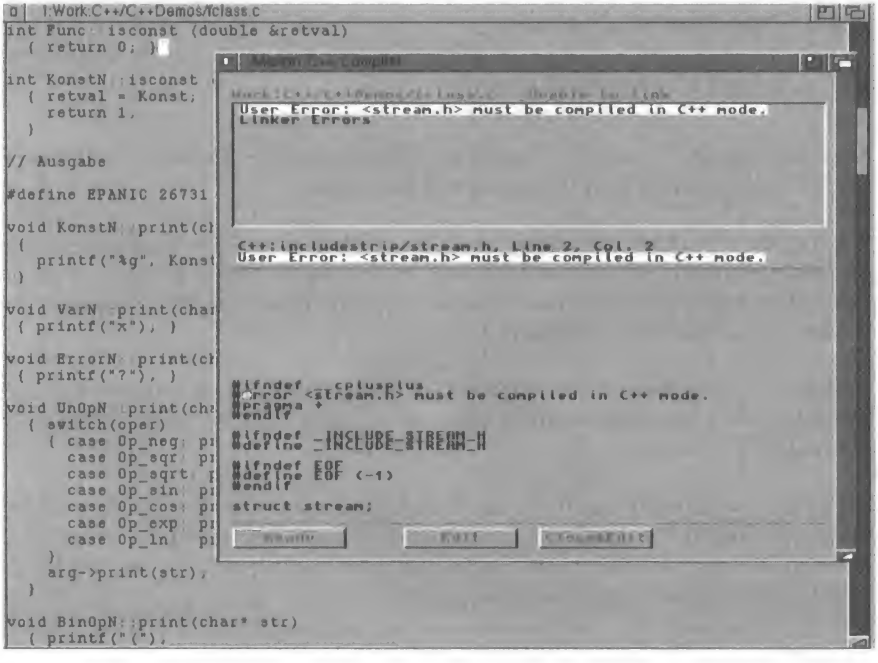
- „Linking": Der Linker bindet das erzeugte Programm mit den nötigen Bibliotheksfunktionen zusammen.

- „Ready": Das Programm ist übersetzt und ausführbar. Wurden keine Fehler oder Warnungen ausgegeben, verschwindet das Fenster bei dieser Meldung automatisch.

Natürlich kann dabei beliebig viel schief gehen, beispielsweise könnte das Programm fehlerhaft sein. Fehlermeldungen und Warnungen aller Art werden in dem Rahmen unterhalb der Statuszeile ausgegeben. „MaxonC++" gibt immer höchstens so viele Fehler

aus, wie in diesen Rahmen passen (je nach Windowgröße zwischen zwei und zehn Stück) und wartet auf eine Reaktion des Benutzers, falls die Höchstzahl erreicht wurde.

Eine solche einzeilige Fehlermeldung allein ist meist natürlich nicht aussagekräftig genug. Deshalb kann sich der Programmierer jeweils zu einer Meldung ausführlichere Informationen anzeigen lassen.



Der Bereich zwischen dem Fehlermeldungs-Rahmen und der unteren Symbolleiste wird im Fehlerfalle durch waagerechte Leisten zweigeteilt. In den beiden oberen Zeilen stehen Quelltextdatei, Zeilen- und Spaltennummer des gerade angewählten Fehlers und noch einmal eine Wiederholung der schon im Rahmen ausgegebenen Fehlermeldung. Unterhalb der horizontalen Linie sehen Sie einen Ausschnitt aus dem Quelltext mit einem Pseudo-Cursor an der fraglichen Position.

Wenn allerdings der Linker Fehler meldet, steht anstelle der Sourceposition (was weiß ein Linker schon von Quelltexten!) schlicht „Linker Error“, und dort, wo sonst der

Quelltext-Ausschnitt dargestellt wird, steht eine eventuell verkürzte Liste der beim Zusammenbinden aufgetretenen Fehler, also hauptsächlich Namen von undefinierten oder gleich mehrfach definierten Linker-Symbolen.

Weniger häufig treten beim Übersetzen unter dem Titel „Fatal Error" Fehler bei der Codegenerierung oder beim Assemblieren auf. Meist sind diese auf Speichermangel zurückzuführen, und dementsprechend heißen sie „Out of Memory", „Workspace overflow".

Last not least gibt es ganz unten im Compiler-Fenster noch eine Symbolleiste. Zunächst finden Sie hier nur ein Symbol mit der Aufschrift „Break". Bei Bedarf werden aber weitere Symbole hinzugefügt.

2.4.1.2 Abbruch und Ausstieg

Im Gegensatz zu einfachen Kommandozeilen-Compilern besitzt das „MaxonC++"-Fenster eine Vielzahl von Knöpfen und Funktionen, an denen der C++-Programmierer herumspielen kann. Alle Symbol-Funktionen sind, um den Bedürfnissen der „echten Programmierer" entgegenzukommen, auch durch Tasten oder Tastenkombinationen bedienbar.

Beim Start des Compilers können Sie im Compiler-Fenster zunächst nur zwei Dinge tun: Das Fenster schließen oder die Übersetzung abbrechen.

Mit einem Klick auf das Schließfeld wird die Übersetzung abgebrochen und das Fenster geschlossen. Die Taste „ESC" hat übrigens die gleiche Wirkung.

Außerdem gibt es unten rechts das bereits erwähnte „Break"-Symbol. Man benutzt es, wenn man den Compiler zwar abbrechen, aber noch irgendwelche im Fenster ausgegebenen Meldungen in Ruhe lesen möchte. Nach einem Klick auf „Break", aber auch dann, wenn der Compiler fertig ist und Ihnen noch Zeit zur Lektüre der Fehlermeldungen geben möchte, verwandelt sich die Symbol-Beschriftung in „Ready", was in diesem Fall keineswegs unbedingt heißt, daß der Compiler fertig ist (das wird ja durch das „Ready" oben rechts in der Statuszeile angezeigt), sondern lediglich Sie dazu ermuntern soll, zu bestätigen, daß Sie fertig sind. Nach einem Klick auf das „Ready"-Symbol wird ebenfalls das Fenster geschlossen.

Auch für das „Break"- bzw. „Ready"-Symbol müssen Sie natürlich nicht zur Maus greifen. Es reicht, wenn Sie hier die kanonische Abbruch-Tastenkombination, nämlich Ctrl-C, drücken.

2.4.1.3 Auswahl von Fehlermeldungen

Laut Murphy geht bei so ziemlich jedem Compiler-Lauf etwas schief, und meist sogar etwas mehr als nur „etwas". Deshalb bietet das Compiler-Fenster Platz für bis zu zehn Fehlermeldungen auf einmal.

Eine dieser jeweils einzeiligen Meldungen ist immer invertiert und somit hervorgehoben, und zu genau dieser Meldung werden unterhalb des Rahmens zusätzliche Informationen angezeigt. Mit einem Mausklick auf eine Fehlerzeile macht man diese zur gerade privilegierten, invertierten Meldung.

Das geht aber auch über die Tastatur: Mit der Leertaste durchläuft man die Fehlermeldungen von oben nach unten, und mit „Backspace" in die andere Richtung.

2.4.1.4 Vom Compiler in den Editor

Der besondere Witz an einer integrierten Umgebung ist ja das perfekte Zusammenspiel von Compiler und Editor, insbesondere das automatische Anspringen von Fehlerstellen.

Das funktioniert in „MaxonC++" denkbar einfach: Sobald eine Fehlermeldung im Compiler-Fenster ausgegeben wird, erscheinen zwei Symbole mit der Aufschrift „Edit" bzw. „Close&Edit". Nun wählen Sie wie oben beschrieben Ihre Wunsch-Fehlermeldung aus (das geht übrigens auch, während der Compiler noch arbeitet) und klicken „Edit" an. Prompt lädt Edward die zugehörige Quelltextdatei (falls der Text nicht ohnehin schon im Editor vorliegt) und setzt den Cursor an die bemängelte Position. Nun können Sie den Fehler berichtigen - vorausgesetzt natürlich, der Compiler läuft nicht noch und hat ein „Lock" auf diesen Text eingerichtet. Das Compiler-Fenster wird dabei nicht geschlossen, so daß Sie noch weitere Fehlerstellen anspringen können.

Die Integration von Compiler und Editor geht allerdings nicht so weit, daß im Compiler-Fenster Quelltext-Veränderungen beachtet würden. Wenn Sie z. B. zuerst einen Fehler berichtigen und dabei einige Zeilen einfügen oder löschen müssen, merkt „MaxonC++" nichts davon. Sollten Sie also anschließend eine weiter unten im Quelltext gelegene Fehlerstelle anspringen lassen, landen Sie gnadenlos in der Zeile mit der Nummer, in der der Fehler lag, als er bemerkt wurde - und durch die inzwischen vorgenommenen Änderungen stimmt diese Zeilennummer vielleicht nicht mehr, und Sie landen einige Zeilen zu hoch oder zu tief. Das dürfte aber in der Praxis kaum ein Problem darstellen, denn im Compiler-Fenster wird immer noch der ursprüngliche Quelltext mit dem Cursor an der richtigen Stelle angezeigt, und durch einen kurzen Textvergleich zwischen Compiler- und Editorfenster können Sie schnell die wirkliche Fehlerposition „zu Fuß" ansteuern. Außerdem ändert man in der Praxis bei Compilerfehlern meist sowieso nur die fragliche Zeile, ohne großartiges Löschen und Einfügen von Zeilen. Das Symbol „Edit" ist mit dem Hotkey „E" belegt.

Manchmal tritt während der Übersetzung nur ein einziger Fehler auf, der vielleicht noch den einen oder anderen Folge-Fehler nach sich zieht, aber trotzdem nur eine einzige Änderung am Quelltext notwendig macht. Deshalb gibt es auch noch das Symbol „Close Edit". Hier wird im Editor die gerade angewählte Fehlerposition angesprungen (genau wie beim „Edit"-Gagdet) und gleichzeitig das Compiler-Fenster geschlossen - wobei natürlich auch der Compiler abgebrochen wird, sollte er noch am arbeiten sein.

Alternativ können Sie auch die „Return"-Taste drücken oder mit der Maus einen Doppelklick auf die Fehlermeldung machen.

2.4.1.5 „Continue" und „Run"

Bei passender Gelegenheit taucht in der unteren rechten Ecke des Compiler-Fensters ein Symbol auf, das entweder mit „Continue" oder mit „Run" beschriftet ist und dementsprechend auch mit dem Hotkey „C" oder „R" belegt ist.

Treten beim Übersetzen bzw. Linken mehr Fehler auf, als im Rahmen dargestellt werden können, wartet der Compiler erst einmal ab, und es erscheint das „Continue"-Symbol. Nach einem Klick darauf vergißt „MaxonC++" die bisher gemeldeten Fehler bis auf den letzten und macht munter weiter. Meist ist es aber nicht unbedingt sinnvoll, nach mehr als zehn Fehlern noch weiter zu übersetzen, denn normalerweise kommt dabei nicht mehr viel sinnvolles ‚raus. Deshalb wird man in der Regel das „Continue"-Symbol meiden und lieber die Fehler korrigieren.

Wenn „MaxonC++" bei der Übersetzung Fehler oder Warnungen ausgegeben hat, aber trotzdem ein lauffähiges Programm erzeugen und linken konnte, taucht unten rechts das „Run"-Symbol auf. Mit einem beherzten Klick darauf können Sie das Programm starten - aber vorher sollten Sie sich genau überlegen, ob die gemeldeten Fehler beim Starten des Programms nicht zu einem Programmabsturz oder ähnlich unangenehmen Effekten führen können!

2.4.2 Ausführen und Abbrechen

Die zweite wichtige Funktion aus dem „Compiler"-Menü heißt „Ausführen" und tut auch genau das: Ein C++-Programm wird gestartet.

Vorher prüft „MaxonC++" allerdings noch, ob nicht vielleicht erst noch der Compiler konsultiert werden muß, und startet selbigen automatisch.

Anschließend wird ein Text-Fenster für die Ein- und Ausgaben des Programms geöffnet. Falls das Programm tatsächlich etwas ausgegeben hat, erscheint nach erfolgreicher Programmausführung die Aufforderung „Press Return“ - Sie sollen also die „Return“-Taste drücken und so bestätigen, daß Sie die Ausgaben des Programms auch gelesen haben.

Während das Programm läuft, sind verschiedene Funktionen des Systems gesperrt, z.B. kann man es nicht ein zweites Mal starten. Man kann den Compiler nicht laufen lassen, weil man dem laufenden Programm ja dadurch den Code unter den Füßen wegziehen bzw. es überschreiben würde. Das ist sicher sinvoll und stört auch nicht weiter - es sei denn, das Programm ist irgendwo hängengeblieben und läßt sich nicht normal beenden.

Gegen „normale” Endlosschleifen hilft die Compileroption „Unterbrechen”. Schwieriger wird es dagegen, wenn der Prozeß abgestürzt ist, auf ein Ereignis wartet, daß nie eintreten kann oder die erwähnte Unterbrechnungs-Option einfach nicht benutzt wurde. Deshalb (und nur für solche Fälle!) gibt es im Compiler-Menü die Funktion „Abbrechen”, die den gestarteten Prozeß schlicht abtötet und dann das Ausgabefenster schließt. Auf den ersten Blick eine praktische Lösung, aber leider gibt es auf dem Amiga (im Gegensatz z.B. zu UNIX) keine saubere Möglichkeit, einen Prozeß aus dem System zu entfernen. Insbesondere werden dabei keine Resourcen (reservierter Speicher, geöffnete Dateien und Fenster etc.) freigegeben. Sie müssen davon ausgehen, daß Sie auf diese Weise freien Speicher verlieren, wenn Sie auf diese brutale Art abbrechen. Aber das ist im Zweifel immer noch angenehmer, als das „MaxonC++”-System neu zu starten oder gar den Rechner zu booten.

2.4.3 übersetzen von Dateien

Als kleines Schmalkerl gibt es auch noch die Menü-Funktion „Compile Datei“, mit der Sie einen Quelltext übersetzen können, ohne ihn in den Editor laden zu müssen. Mit dem Dateiauswahlfenster werden Sie aufgefordert, die Textdatei auszuwählen. Im Falle eines Fehlers stehen Ihnen alle Möglichkeiten des Fehlerstellenansprungs zur Verfügung. Wenn das Programm übersetzt und gelinkt werden konnte, können Sie es ganz wie ein aus dem Editor compiliertes Programm ausführen oder sich Objekt- bzw. Exe-Dateien erzeugen lassen.

2.4.4 Exe- und Objektdateien

Ungeachtet allen damit verbundenen Komforts werden Sie Ihre Programme wohl nicht immer nur aus der integrierten Umgebung starten wollen. Deshalb gibt es im „Compiler“-Menü zwei Funktionen, die das übersetzte Programm in Dateien schreiben:

- „Exe-Datei" erzeugt eine ausführbare Programmdatei, vorausgesetzt natürlich, ein Programm ist korrekt übersetzt und gelinkt worden.

- „Objekt-Datei" schreibt zur zuletzt compilierten Übersetzungseinheit eine Objektdatei, aus der man später z. B. mit dem „CPPC" ein ausführbares Programm erzeugen kann. Im „Make"-Modus ist diese Funktion nicht unbedingt sinnvoll, denn dort werden diese Objektdateien ja beim Compilieren automatisch erzeugt.

2.4.5 Daten und Informationen

Mit dem Menüpunkt „Information" öffnet man ein Fenster, in dem Informationen über den Status von „MaxonC++" angezeigt und ständig aktualisiert werden. Man schließt das Info-Fenster wieder, indem man das Schließsymbol anklickt oder im aktivierten Info-Fenster eine beliebige Taste drückt.

Welche mehr oder weniger hilfreichen Informationen angezeigt werden, hängt von der aktuellen Windowgröße ab. Bei der normalen Größe, wie Sie sie vorfinden, wenn Sie das Fenster öffnen, sehen Sie einen repräsentativen „MaxonC++"-Schriftzug, eine kurze Information über Urheberschaft und Copyright und einige Daten, auf die im folgenden eingegangen werden soll. Wenn Sie das Info-Fenster noch etwas vergrößern, finden Sie etwas ausführlichere Angaben über „MaxonC++" und Maxon, und wenn Sie es hinreichend verkleinern (aber natürlich nicht so klein machen, daß nichts mehr hineinpaßt), stehen dort nur noch die geheimnisvollen Kürzel und Daten.

Ganz allgemein gesagt, stehen dort (falls da überhaut etwas steht, nämlich nur dann, wenn ein compiliertes Programm vorliegt) jeweils von „MaxonC++" Speicherbereiche mit ihren Anfangs- und Endadressen, ihrer Größe und natürlich ihrer Bedeutung.

Sie können dort folgende Daten finden:

Exe Gesamtgröße des ausführbaren Programms, so weit es im FAST-Mem liegt.

Wenn es einen Hunk namens Chip gibt, umfaßt dieser den zusätzlichen Speicherplatz im CHIP-RAM.

Free Der Teil des Arbeitsspeichers („Workspace"), der nicht mit dem Executable ausgefüllt ist.

Code Codehunk des soeben übersetzten Moduls

Data Hunk für initialisierte Daten des Moduls

Bss Speicherbereich für nicht initialisierte Daten

Dt.c Chip-RAM-Daten

SDta Datenhunk für „Small Data". Genau wie „Code", „Data" und „Bss" bezieht sich auch diese Angabe nur auf die jeweils zuletzt compilierte Übersetzungseinheit und ist deshalb im „Make"-Modus nicht so besonders erbaulich.

Natürlich werden nicht immer alle diese Daten ausgegeben, sondern nur die, welche gerade real existieren. Um die Verwirrung komplett zu machen, wird nach erfolgreichem Linken auch noch die „Maximum code size" ausgegeben, also noch ein Maß für den Umfang des erzeugten Codes, aber auch das ist noch nicht genau die spätere Größe einer ausführbaren Programmdatei. Das Ganze hängt folgendermaßen zusammen:

Ein ausführbares Programm besteht aus Daten, die im Chip-Memory liegen müssen (insbesondere Grafikdaten) und solchen, die in einen beliebigen Speicherbereich geladen werden dürfen, vorzugsweise natürlich ins Fast-RAM. Diese beiden Datenbestände werden durch die Angaben „Exe" und „Chip" repräsentiert und ihre Summe ist immer genau die „Maximum code size". Das Wort „maximum" können Sie dabei vorerst getrost vergessen, denn in der vorliegenden Version beherrscht MaxonC++ noch keine Overlays, und der Speicherplatzbedarf ist deshalb eine konstante Größe. Diese Angabe bezieht sich aber auf die Anzahl der Bytes, die das Programm belegt, wenn es in den Speicher geladen wurde, wogegen bei der ausführbaren Programmdatei noch ein gewisser Overhead dazukommt.

Ein zusammengelinktes Programm ist aber kein amorpher Klumpen von Bytes, sondern durch „Hunks" strukturiert. Jede Übersetzungseinheit (C-typische Bezeichnung für „Modul") eines Programms besteht aus wenigstens einem Hunk, nämlich typischerweise einem Code-Hunk für den ausführbaren Code, eventuell je einem Data- oder Bss-Hunk für die initialisierten bzw. uninitialisierten statischen Daten oder alternativ einem gemeinsamen, Adressregister-relativ adressierten Datenhunk (beim „Small Data"-Modell) sowie manchmal einem zusätzlichen Hunk für Daten, die in das Chip-Memory gehören.

MaxonC++ gibt auch die Größe dieser Hunks (soweit vorhanden) aus, aber nur die Hunks des zuletzt übersetzten Moduls. Hinzu kommen stets noch einige Hunks für die Bibliotheken und bei Projekten, die aus mehreren Übersetzungseinheiten bestehen, natürlich die Hunks der übrigen Module. Deshalb ist die Summe der ausgegebenen Hunks nicht identisch mit der Gesamt-Codegröße des Programms.

2.5 Features und Optionen

2.5.1 Ein kurzer Überblick

Für viele Programmierer ist das Schönste an einem Compiler, daß man so viele Optionen und Switches hat, an denen man herumspielen kann. Bei „MaxonC++“ nimmt man sämtliche dieser Einstellungen über Dialogfenster vor, deren es genau drei gibt: Eines („System“) für allgemeine Einstellungen, die in irgendeiner Weise mit dem „MaxonC++“-System zu tun haben, ein weiteres für die unvermeidlichen Compiler-Optionen und schließlich einen eigenes Dialogfenster für die Linker-Steuerung.

Wie Sie sich vielleicht denken können, beherrschen „MaxonC++“ und die Kommandozeilen-Version „CPPC“ praktisch dieselben Einstellungen und Optionen, nur die Bedienung ist eben unterschiedlich. Deshalb sollten Sie sich dieses Kapitel auch dann durchlesen, wenn Sie ausschließlich den „CPPC“ benutzen wollen, denn bei der Beschreibung der Optionen im „CPPC“-Kapitel wird hauptsächlich das „Wie“ der Einstellungen beschrieben, während nur hier ausführlich erläutert wird, was diese Optionen genau bedeuten.

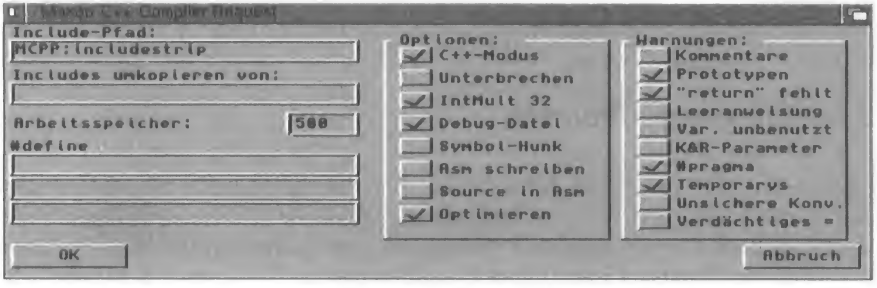
Zurück zum „MaxonC++“: Wenn Sie wollen, können Sie alle drei Dialogfenster gleichzeitig öffnen. In jedem der Fenster finden Sie am unteren Rand die obligatorischen Symbole „OK“ und „Cancel“, mit denen Sie die vorgenommenen Einstellungen bestätigen bzw. sie wieder verwerfen können.

2.5.2 Compiler-Optionen

2.5.2.1 Include-Dateien

Wie eigentlich alle modernen Programmiersprachen, sind C und C++ ziemlich „dumm“: Es gibt für den Compiler keinerlei vordefinierten Funktionen (während z. B. in Pascal „WriteLn“ oder in BASIC „print“ stets zur Verfügung stehen). Vielmehr stehen Definitionen für die Standard-Funktionen in diversen Includedateien, die man explizit einbinden muß.

Vielleicht wissen Sie schon, daß es dafür zwei verschiedene Möglichkeiten gibt: Schließt man den Dateinamen „so“ in Anführungszeichen ein, sucht der Compiler (oder der Preprozessor, um genauer zu sein) vom aktuellen Verzeichnis aus, während die <so> in spitze Klammern gesetzten Dateinamen Standard-Includedateien bezeichnen und in einem ganz bestimmten Verzeichnis gesucht werden. Von genau diesem Verzeichnis, und wie man es festlegt, soll hier die Rede sein.



Im „Compiler"-Dialogfenster legen Sie mit den beiden Texteingabefelder oben links eben dieses Verzeichnis fest. Wenn Sie nur das obere Symbol benutzen, dürfte die Funktionsweise klar sein: Der Compiler lädt die Include-Dateien aus dem dort angegebenen Verzeichnis. Als kleines Bonbon vor allem für Diskettenbenutzer gibt es aber auch noch die Möglichkeit, die benutzten Dateien auf einen schnelleren Datenträger umkopieren zu lassen: Wenn Sie einen doppelten Suchpfad angeben, sucht MaxonC++ zuerst unter dem zuerst genannten Pfad, der z. B. in der RAM-Disk liegen könnte. Findet er die Datei dort nicht, versucht er es über den zweiten Pfad, z. B. auf einer Diskette, und kopiert die Datei in das erste Verzeichnis um.
Beim nächsten Übersetzungsvorgang liegt die Datei dann auf dem schnellen Medium vor. Übrigens legt MaxonC++ vor dem Umkopieren bei Bedarf selbstständig die benötigten Unterverzeichnisse an.

Vor einer kleinen Heimtücke des Systems sei noch gewarnt: Achten Sie bitte darauf, daß Sie weder an den Anfang noch an das Ende der Dateinamen versehentlich irgendwelche Leerzeichen anhängen! „MaxonC++" versucht sonst gänzlich ungerührt, auf die angegebenen Pfade einschließlich der überflüssigen Spaces zuzugreifen, und da wird er naturgemäß nicht allzu viel finden.

2.5.2.2 Der Arbeitsspeicher des Compilers

Hin und wieder braucht der MaxonC++ Compiler einen zusammenhängenden Speicherbereich, nämlich bei Codegenerierung und Assemblierung. Außerdem legt er nach erfolgreichem Linken in diesem Speicherblock eine lauffähige Kopie des erzeugten Programms ab, und sollte dabei noch genug übrig bleiben, benutzt er den Speicher für den Stack, wenn das Programm aus der „MaxonC++"-Umgebung gestartet wird.

Was Sie damit zu schaffen haben? Eigentlich nichts, aber leider weiß „MaxonC++" nicht im Voraus, wieviel Speicher er dafür brauchen wird. Der Compiler kann später höchstens einen Überlauf feststellen, aber vergrößern kann er diesen „Workspace" genannten Speicherblock dann nicht mehr, weshalb er mit einer Meldung wie „Workspace Overflow" aussteigt.

Deshalb müssen Sie leider persönlich festlegen, wie groß dieser Workspace sein soll, und zwar mit den entsprechendem Integer-Symbol im Compiler-Dialogfenster. Der Defaultwert von 80 kByte dürfte übrigens für die allermeisten Zwecke ausreichen, und wenn Sie mit RAM etwas knapp sind, können Sie ruhig versuchen, die Workspace-Größe weiter zu verringern. Sollte der Compiler dagegen einmal „Workspace Overflow" melden, setzen Sie die Größe einfach etwas herauf und versuchen es noch einmal.

2.5.2.3 Vordefinieren von Symbolen

Mit den drei Symbolen, die mit „#define" überschrieben sind, können Sie bis zu drei Preprozessor-Symbole vordefinieren. Jeder Eintrag in einem dieser Texteingabefelder wirkt dann so, als stünde er mit einem vorangestellten „#define" am Anfang jedes übersetzten Quelltexts. Dieses Feature entspricht also der „-d"-Option von „CPPC".

Ein Beispiel: Sie haben als vorsichtiger Programmierer die Includedatei „<assert.h>" eingebunden und diverse Sicherheitsabfragen in Ihr Programm gesetzt - (übrigens ein löbliches Unterfangen). Um diese Abfragen automatisch aus dem Programm auszublenden, müssen Sie dann „#define NDEBUG" irgendwo an den Anfang des Quelltexts setzen oder ganz einfach „NDEBUG" in eines der drei „#define"-Symbole schreiben.

Sie können Symbolen aber nicht nur „einfach so" definieren, sondern ihnen auch Werte zuweisen. Ein Symbol-Eintrag wie „N 100" definiert das Symbol „N" als „100". Leider akzeptiert der Compiler bei solchen Vorab-Definitionen immer nur ein Token pro Symbol. Der Eintrag „ANZAHL (17+4)" belegt „ANZAHL" mit dem Zeichen „(„ und ignoriert den Rest einfach. Aber komplexe Definitionen wird man normalerweise ohnehin in den Quelltext schreiben, während derartige „externe" Symboldefinitionen eher für irgendwelche Flags wie „NDEBUG" benutzt werden.

2.5.2.4 Diverse kleine, aber trotzdem wichtige Schalter

Nahezu unübersehbar befinden sich im Compiler-Dialogfenster zwei Spalten mit diversen Switches, also so etwas wie ein virtuelles Mäuseklavier. Die linke Spalte ist mit „Optionen" überschrieben und enthält demzufolge eine mehr oder weniger geordnete Ansammlung von ein- und ausschaltbaren Compileroptionen.

2.5.2.4.1 C++

Mit dem Symbol „C++" können Sie wählen, ob MaxonC++ als echter C++-Compiler oder kastriert im ANSI C-Modus arbeiten soll. Ich hoffe doch stark, daß Sie die C++-Option

nur in Ausnahmefällen ausschalten werden. Sie können diese Umschaltung übrigens auch mit einer „#pragma"-Zeile direkt im Quelltext vornehmen. Näheres dazu erfahren Sie im Kapitel über den Preprozessor.

2.5.2.4.2 Unterbrechen

Mit „Unterbrechen" legen Sie fest, ob der Compiler im erzeugten Code Breakpoints setzen soll. Wenn diese Option angewählt ist, baut er in jede Schleife eine Abfrage der Tastenkombination „Ctrl-C" ein.

2.5.2.4.3 IntMult 32

Der 68000er-Prozessor kann leider keine Langworte multiplizieren und dividieren, sondern nur zwei 16-Bit-Werte zu einem 32-Bit Ergebnis multiplizieren und eine 32-Bit-Zahl durch ein 16-Bit-Wort dividieren, wobei Quotient und Rest jeweils 16 Bit breit sind. Für numerische Anwendungen ist das natürlich nicht so schön, weshalb MaxonC++ optional solche Operationen durch Bibliotheksfunktionen ersetzen kann.

Diese korrekt rechnenden Funktionen werden benutzt, wenn das Symbol „IntMult 32" aktiviert ist, und kosten natürlich einiges an Rechenzeit. Dieser Laufzeit-Aufwand ist aber in der Praxis nicht so gravierend, und für korrekte Ergebnisse ist uns schließlich nichts zu teuer.

2.5.2.4.4 Debug-Datei

Die Compileroption „Debugger-Datei" ist für Sie nur von Interesse, wenn Sie den Maxon Debugger besitzen. Der Compiler legt dann zu jeder Übersetzungseinheit eine Datei an, aus welcher der Debugger umfangreiche Informationen entnehmen kann.

2.5.2.4.5 Symbol-Hunk

Anstelle eines Source-Level-Debuggers wie dem hochkomfortablen Maxon Debugger (MDB) kann man natürlich auch einen symbolischen Debugger wie „MetaScope" benutzen. Um dabei überhaupt etwas wiederzufinden, sollte man sich bei Exe- und Objektdateien sogenannte „Symbol-Hunks" erzeugen lassen. Genau dafür dient die gleichnamige Option.

Wichtig ist, daß Sie „Symbol-Hunk" schon beim Compilieren und nicht erst beim Erzeugen der Exe-Dateien anwählen, um den Compiler zu veranlassen, die notwendigen Informationen zu sammeln und aufzubewahren.

2.5.2.4.6 Asm schreiben

Wenn „Asm schreiben" angewählt ist, erzeugt der Compiler zu jedem Text, den er erfolgreich übersetzt, ein Assembler-Listing. Der Assembler-Quelltext wird in eine Datei geschrieben, deren Name wie der des C-Quelltexts anfängt und mit „.s" endet, also z. B. „hallo.c" zu „hallo.s".

Wenn Sie schon andere Compiler näher kennen, wundern Sie sich vielleicht darüber, daß der Compiler scheinbar einen so optimalen Assembler-Text erzeugt, daß für den Assembler nichts mehr zum Optimieren übrig bleibt. Auch werden Sie sich vielleicht fragen, warum Sie mit dem integrierten Assembler von MaxonC++ nicht selbstgeschriebene Assembler-Programme übersetzen lassen können.

In beiden Fällen handelt es sich um Fehlschlüsse, denn MaxonC++ ist etwas ungewöhnlich implementiert. Der Compiler erzeugt überhaupt keinen Assembler-Quelltext, und es gibt auch keinen Assembler.

Um diese kryptische Bemerkung etwas zu erhellen: Der Compiler erzeugt bei der Codegenerierung einen Zwischencode in einem internen binären Format, das zwar in gewisser Weise wie ein Assembler-Quelltext aufgebaut ist, sonst aber absolut nichts mit einer ASCII-Textdatei zu tun hat. Der „Assembler" ist dann ein ziemlich dröges kleines Prograämmchen, daß diesen Zwischencode in echten Maschinencode übersetzt und dabei noch ein paar kleine Optimierungen durchführt. Diese Low-Level-Optimierungen, zu denen z. B. Short Branches gehören, trägt der Assembler dann auch immer gleich im Zwischencode ein, damit er beim nächsten Pass dasselbe nicht noch einmal optimieren muß.

Und dann gibt es noch einen Programmteil, der noch viel simpler als der „Assembler" ist und den internen Zwischencode in lesbaren Assembler-Quelltext umwandelt. Diese Funktion wird benutzt, wenn die Option „Asm schreiben" angewählt ist, und da die Datei geschrieben wird, nachdem der Pseudo-Assembler seine Arbeit getan hat, sind darin schon alle Optimierungen vorgenommen. Der von MaxonC++ generierte Assembler-Source entspricht also genau dem erzeugten Code. Sie können sich den Reassembler sparen, wenn sie die Codequalität genauer unter die Lupe nehmen wollen.

2.5.2.4.7 Source in Asm

Den oben beschriebenen Quelltext kann man sich etwas übersichtlicher gestalten lassen, indem man zusätzlich die Option „Source in Asm" wählt. Dann wird nämlich der C-Quelltext an jeweils passender Stelle in den Assembler-Source als Kommentar eingefügt, so daß sich die Bedeutung der endlosen Assembler-Listings dem menschlichen Leser einfacher erschließt.

2.5.2.4.8 Optimieren

Guter Code ist mit einem gewissen Aufwand verbunden. Sowohl in Bezug auf den Speicherplatzbedarf des Compilers als auch auf die Übersetzungszeit. Deshalb gibt es die wahlweise ab- oder zuschaltbare Option „Optimieren", die MaxonC++ veranlaßt, die Codeerzeugung etwas schlauer aufzuziehen.

Das Thema „Optimieren" ist nahezu unerschöpflich, und folglich dürfte bei jeder neuen Version von MaxonC++ das eine oder andere Feature hinzukommen. Im Moment macht der Compiler bei aktiver Optimier-Option mindestens folgendes:

- Alle anderweitig entbehrlichen Register werden für lokale Variablen benutzt. Die Entscheidung, welche Variablen in Register geladen werden sollen, fällt über gewichtetes Zählen ihres Auftretens.

- Wenn sich herausstellt, daß eine Funktion keine lokalen Variablen auf dem Stack mehr benötigt (z. B. weil alle in Register umgelegt werden konnten), wird das Einrichten eines Stack-Rahmens mittels „LINK" und „UNLK" unterdrückt.

- Der Assembler führt nicht nur einen, sondern zwei optimierende Passes durch. Damit werden z. B. auch Sprünge nach vorn zu Short-Branches optimiert.

Diverse Mini-Optimierungen, die keinen besonderen Aufwand erfordern, werden natürlich immer durchgeführt und sind nicht abschaltbar. Die Grenze zwischen „vermiedenem schlechtem Code" und einer „Optimierung" ist bekanntlich fließend.

2.5.2.5 Optionale Warnungen und Meldungen

Diverse Sprachkonstrukte sind zwar an sich syntaktisch wie semantisch korrekt, aber trotzdem etwas obskur und möglicherweise Folge eines Versehens des Programmierers. Deshalb gibt es zehn Warnungen, die man je nach Situation und persönlichem Programmierstil einzeln ein- und ausschalten kann. In „MaxonC++" dienen dazu die Switches in der rechten Spalte des Compiler-Dialogfensters.

Im einzelnen gibt es die folgenden Warnungen:

- „Kommentare": Bei verschachtelten Kommentaren, wie „/* Test /* Nahu? */" wird gewarnt. Hier liegt der Verdacht nahe, daß am Ende eines Kommentars einfach das „*/" vergessen wurde.

- „Prototypen": Im C-Modus wird gewarnt, wenn eine Funktion aufgerufen wird, zu der kein Prototyp deklariert wurde. In C++ ist das bekanntlich sowieso ein Fehler.

- „Return' fehlt": Wenn in einer Funktion, deren Ergebnistyp nicht „void" ist, nirgendwo ein „return" auftaucht, ist sicher etwas faul. Deshalb sollte diese Warning immer eingeschaltet sein.

- „Leeranweisung": Eine Anweisung wie „42;" hat keinerlei Wirkung und war folglich wahrscheinlich nicht so gemeint. Ein besonders beliebter Fehler ist auch der Aufruf einer parameterlosen Funktion ohne Argumentliste, etwa „test;" statt „test();".

- „K&R-Parameter": Das „K&R" steht für „Kernighan & Ritchie" und damit für den Vor-ANSI-Standard. Mit dieser Option kann man sich bei Parameterlisten im alten Stil eine Warnung ausgeben lassen.

- „#pragma": Es ist natürlich Sinn und Zweck eines „#pragma", daß der Compiler darin eben keine Fehler meldet und alles ignoriert, was er nicht kennt. Wenn Sie aber nicht bloß fremde Quelltexte übernehmen, sondern mit MaxonC++ selbst entwickeln, ist es sicher sinnvoll, wenn Sie sich auf unbekannte Pragma's hinweisen lassen, schließlich könnte es sich dabei ja auch ganz einfach um einen Tippfehler handeln.

- „Temporarys": Laut C++-2.0-Standard ist es ein Fehler, wenn eine Referenz auf einen nicht-konstanten Typ mit einem Nicht-L-Wert initialisiert wird und dabei ein temporäres Objekt eingeführt werden muß. Da diese Regel im 1.0-Standard noch nicht existierte, gibt es dafür keine „echte" Fehlermeldung, sondern eine abschaltbare Warnung.

Ein Beispiel:

```
void dup(int &ir)
{ ir *= 2; }

void main()
{ int i;
  dup(i);        // OK

  long l;
  dup(l);        // VORSICHT!
}
```

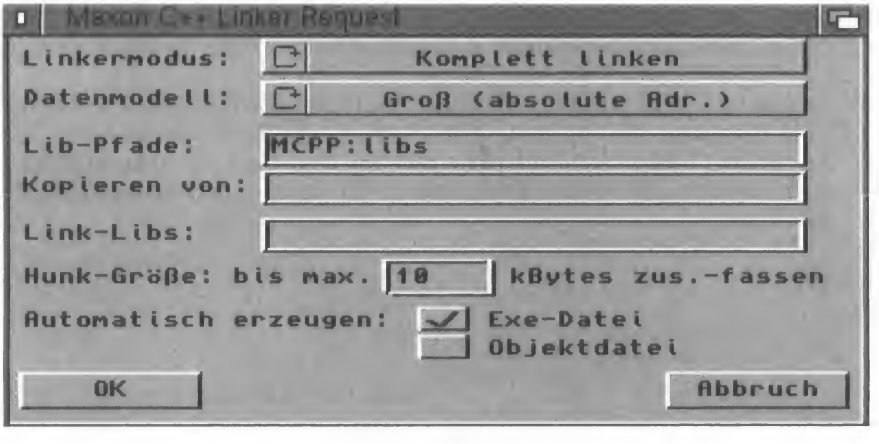
Da hier „l" von „int" nach „long" konvertiert werden muß (in MaxonC++ übrigens eine leere Operation, denn intern sind beides 32Bit-Zahlen), wird ein temporäres Objekt eingeführt, und der Programmierer erlebt höchstwahrscheinlich eine Überraschung, denn eim zweiten „dup"-Aufruf gibt es keinen Seiteneffekt auf das Argument. Die neue C++-Regel ist also ausgesprochen sinnvoll.

- „Unsichere Konv.": Diese Option warnt bei impliziten Typumwandlungen (also ohne Cast) von ganzzahligen in Fließkommatypen.

- „Verdächtiges =": Dabei handelt es sich um das beliebte Konstrukt „if (a=b)..." - korrekt und bei vielen Programmierern beliebt, aber auch ein typischer Anfängerfehler statt „if (a==b)". Wenn die entsprechende Option aktiv ist, wird jeweils eine Warnung ausgegeben, wenn ein „=" auf diese Weise in einem logischen Ausdruck auftritt, also als Bedingung in „if", „while", „for" oder „do" sowie als Operand von „||", „&&" oder „!".

2.5.3 Das Linker-Dialogfenster

Mit dem Menüpunkt „Linker" aus dem „Optionen"-Menü öffnen Sie das Linker-Dialogfenster. Hier finden Sie diverse Optionen zur Steuerung des Linkers.



2.5.3.1 Datenmodelle

Die Umschaltung zwischen „kleinem" und „großem" Datenmodell hat sich eher zufällig in das Linker-Dialogfenster verirrt, da dies eigentlich eher etwas mit der Codeerzeugung und folglich mit dem Compiler zu tun hat.

Zuerst einmal die Frage: Was hat es eigentlich mit diesen Datenmodellen auf sich?

Wenn im vom Compiler generierten Code auf globale, statische Daten zugegriffen werden soll, geschieht dies normalerweise über absolute Adressen. Die Daten selbst können dann irgendwo im Speicher liegen und über beliebig viele Daten-"Hunks" verteilt sein.
Auch die Gesamtgröße dieser Daten ist nur durch die Speichergröße des Rechners begrenzt. Diese kanonische Art der Adressierung bezeichnet man in diesem Zusammenhang als „großes" Datenmodell („Large Data").

Das klappt zwar recht gut und unterliegt auch keinerlei Beschränkungen, aber wie so oft gibt es auch hier Leute, die daran etwas zu mosern haben. Zum Beispiel benötigt eine absolute Adresse im Code immer 32 Bit, also vier Bytes. Das bläht nicht nur die Codegröße auf, sondern kostet auch entsprechend viel Laufzeit. Ferner müssen alle diese Adressen korrigiert werden, wenn das Programm in den Speicher geladen wird, was in der Dateigröße einen gewissen Overhead bedeutet.

Deshalb ist ein anderer Ansatz, das sogenannte „kleine" Datenmodell, in gewisser Hinsicht schlauer. Hier wird ein Prozessorregister, nämlich das Adressregister A4, als Zeiger auf die statischen Daten des Programms benutzt. Die Daten werden jetzt mit einem16 Bit-Offset relativ zu diesem Datenregister adressiert, was jeweils zwei Bytes Code und ein Nichtigstel an Laufzeit spart. Aber jetzt kommen die Nachteile von „Small Data":

- Mit 16 Bit kann man nur 65536 Bytes adressieren - also ist dies die Obergrenze für die globalen Daten des Programms.

- Alle Datenhunks des Programms müssen in einem einzigen Small-Data-Hunk zusammengefaßt werden. Insbesondere unterscheidet man dann nicht mehr zwischen initialisierten Daten („DATA"-Hunk) und den platzsparend abgespeicherten uninitialisierten Daten („BSS-Hunk). Dadurch kann die ausführbare Programmdatei - trotz geringeren Speicherbedarfs des Programms im RAM - überraschend groß werden.

- Bei der Codegenerierung kann das Register A4 nicht für Variablen oder Zwischenergebnisse genutzt werden.

Hinzu kommen noch einige Eigenheiten des Small-Data-Models von MaxonC++:

- Jede Übersetzungseinheit adressiert nur seine „eigenen" Daten relativ zu A4, nicht jedoch irgendwelche importierte Daten. Das hat den Vorteil, daß Code mit „Small Data" und „Large Data" beliebig mischen kann: Von den innerhalb derselben Übersetzungseinheit definierten Daten weiß der Compiler schließlich, ob er sie im „Small Data"-Hunk abgelegt hat, und kann problemlos A4-relativ adressieren. Bei importierten Daten weiß er das nicht und adressiert deshalb im Zweifelsfall absolut. Dieses Konzept schränkt den Nutzen des kleinen Datenmodells bei Programmen, die aus mehreren Modulen bestehen, natürlich etwas ein.

- Das „Small Data"-Konzept von MaxonC++ ist nicht völlig kompatibel zu dem anderer Compiler oder Assembler.

Welche Konsequenz soll man aus diesen Vor- und Nachteilen ziehen? So ganz allgemein wohl gar keine. Im Zweifelsfall sollten Sie einfach ausprobieren, mit welchem Datenmodell Ihr Programm sich besser macht.

Rein praktisch wird das Datenmodell in „MaxonC++" mit dem entsprechenden Symbol im „Linker"-Dialogfenster und beim „CPPC" mit der Option „-gd" bzw. „-gD" eingestellt.

Noch eine Anmerkung zum Schluß: Ich erwähnte oben, daß MaxonC++ bei „large data" das Register A4 munter benutzt. Wie paßt das mit der Aussage zusammen, daß man „small-" und „large-data"-Code beliebig mischen kann? Nun, A4 wird auch bei „großem" Datenmodell nur in solchen Funktionen getrasht, die selbst keine weiteren Funktionen aufrufen und folglich insbesondere keine Funktionen mit A4-relativen Daten. Natürlich wird in solchen Funktionen A4 auch gegebenenfalls auf den Stack gerettet und am Ende der Funktion wieder hergestellt. Damit merkt die gesamte „Umwelt" nichts davon, wenn in einer „large-data-"Funktion A4 mißbraucht wird.

2.5.3.2 Linken oder nicht Linken - das ist hier die Frage

Auch unter „MaxonC++" haben Sie die Möglichkeit, das Linken ganz oder teilweise zu unterlassen, wenn auch nicht mit ganz so vielen Optionen wie beim CPPC. Dazu dient das Symbol „Linkermodus" im Linker-Dialogfenster (wo auch sonst?). Die drei Einstellungen:

- „Komplett linken": Das übersetzte Programm wird ganz normal mit den Bibliotheksroutinen zu einem ausführbaren Programm zusammengebunden oder zumindest wird es versucht.

- „Nicht linken": Es werden keinerlei Bibliotheks- oder sonstige Module hinzugeladen. Der Linker meldet auch keine Fehler, sondern wird überhaupt nicht erst gestartet. Dieser Modus ist recht angenehm, wenn man an einem Executable nicht interessiert ist und lediglich eine Objektdatei haben möchte.

- „Ohne Startup-Code“: Vorsicht, eine Einstellung für Spezialisten - wenn man damit nicht umgehen kann, stürzen die erzeugten Programme gnadenlos ab! Näheres über diese Option dazu finden Sie im Linker-Kapitel, und zwar in Abschnitt 4.6.

2.5.3.3 Bibliotheken

Der Linker von MaxonC++ muß natürlich auch irgendwie wissen, woher er die Standard-Bibliotheken laden muß. Deshalb trägt man die entsprechenden Pfade im „Lib-Pfade“-Symbol des Linker-Dialogfensters ein. Genau wie bei den Includedateien (siehe 2.5.2.1) kann man auch hier bis zu zwei verschiedene Pfade angeben und Dateien dabei automatisch umkopieren lassen.

Von sich aus versucht der Linker immer nur, undefinierte Symbole im „Logfile“ zu suchen und anschließend die entsprechenden Bibliotheksmodule zu laden. Wenn Sie eine Linker-Library, etwa die allseits beliebte „amiga.lib“, einbinden wollen, müssen Sie deren Namen (ohne „.lib“) im Symbol „Link-Libs“ eintragen. Da es nur ein solches Symbol gibt, können Sie darin auch mehrere solcher Bibliotheken durch Leerzeichen getrennt eintragen, z. B. „amiga debug“.

Diese Bibliotheksnamen können wahlweise mit einem vollständigen Pfad oder ganz ohne Pfad angegeben werden. In letzterem Fall sucht der Linker sie wie die „normalen“ Bibliotheksmodule im dafür spezifizierten Verzeichnis.

2.5.3.4 Zusammenfassen von Hunks

Jedes Modul eines Programms besitzt normalerweise einen Code-Hunk und außerdem noch bis zu drei verschiedenen Datenhunks. Bei einem Programm, das aus mehreren Übersetzungseinheiten besteht, kommt da schon so einiges zusammen, einmal ganz zu schweigen von den Bibliotheksmodulen. Damit das mit der Hunkanzahl nicht Überhand nimmt, ist es üblich, gleichartige Hunks zu einem einzigen zusammenzufassen.

Je weniger Hunks ein Programm besitzt, desto kleiner ist die ausführbare Programmdatei, denn jeder Hunk benötigt einen Overhead von ein paar Bytes. Andererseits ist es aber auch nicht schlau, grundsätzlich alle jeweils gleichartigen Hunks zusammenzufassen, denn ein 250 kByte großer Hunk benötigt nun einmal einen ebenso großen, zusammenhängenden Speicherbereich. Bei Rechnern, die knapp an RAM sind, kommt es durchaus vor, daß der Speicher so fragmentiert ist, daß zwar durchaus noch fünf Blöcke zu je 50 kByte, aber kein einziger zusammenhängender 250k-Block frei ist - ein Programm mit einem unangemessen großen Hunk könnte also nicht mehr geladen werden.

Deshalb kann man eine Höchstgrenze wählen, die der Linker beim Kombinieren von Hunks nicht überschreiten soll. Der entsprechende Betrag ist in kByte im entsprechenden Symbol des Linker-Dialogfensters einzutragen. Wenn Sie z. B. aus irgendwelchen Gründen wollen, daß überhaupt keine Hunks zusammengefaßt werden, setzen Sie den Wert einfach auf Null.

In einem Fall nimmt sich der Linker aber auf jeden Fall eine gewisse Freiheit heraus: Bibliotheksmodule werden immer zu Blöcken von 10 kByte kombiniert, auch wenn Sie einen kleineren Wert vorgegeben haben. Das liegt ganz einfach daran, daß die automatisch eingebundenen Bibliotheksfunktionen naturgemäß aus etlichen kleinen Hunks bestehen und es einfach Wahnsinn ist, jeden davon im ausführbaren Programm einzeln auftauchen zu lassen.

2.4.3.5 Schreiben von Dateien

Einer der Vorzüge einer integrierten Umgebung wie bei MaxonC++ besteht darin, daß sich ziemlich alles im RAM und ohne Diskette- bzw. Festplattenzugriffe abspielen kann: Die Quelltexte werden aus dem RAM gelesen und das ausführbare Programm auch dort abgelegt.

Nun gibt es aber bestimmte Gründe dafür, sich stets eine aktuelle ausführbare Programmdatei schreiben zu lassen. Deshalb gibt es im Linker-Dialogfenster ein Symbol das Sie aktivieren können, wenn nach jeder erfolgreichen Übersetzung eine Exe-Datei geschrieben werden soll. Die die Compileroption zum Erzeugen einer Datei für den Maxon Debugger eingeschaltet, so wird diese ebenfalls geschrieben.

Mit einem weiteren Symbol können Sie außerdem Objektdateien automatisch erzeugen lassen. Im Make-Modus geschieht das natürlich sowieso, so daß dieses Symbol dann keine Bedeutung hat.

2.5.4 „MaxonC++"-Systemeinstellungen

Mit dem dritten Dialogfenster aus dem „Optionen"-Menü können diverse Einstellungen des „MaxonC++"-Systems vorgenommen werden.

2.5.4.1 Autosave

Wenn die „Autosave"-Option vom „MaxonC++" eingeschaltet ist, werden beim Start eines Programms aus der integrierten Umgebung automatisch alle Editor-Texte ab-

gespeichert, sofern dies noch nicht geschehen ist - bestimmt eine sinnvolle Maßnahme, zumal man in C stets ohne Netz und doppelten Boden programmiert. Dieser Autosave sollte nicht mit dem zeitabhängigen Autosave des Edward verwechselt werden.

2.5.4.2 Erzeugen von Icons

„Icons" legt fest, ob MaxonC++ beim Schreiben von ausführbaren Dateien, Makefiles und Debugger-Dateien jeweils ein Icon erzeugen soll.

2.5.4.3 „MaxonC++" - ein polyglottes System

MaxonC++ ist zweisprachig konzipiert. Den größte Teil der Benutzeroberfläche von „MaxonC++" kann man zwischen Deutsch und Englisch umschalten, lediglich der Compiler und seine Fehlermeldungen wurden in Englisch gehalten. Mit dem Gagdet „Sprache" kann die gewünschte Sprache ausgewählt werden.

2.5.4.4 Protokollieren von Ein- und Ausgaben

Wenn Sie die Option „Protokoll" einschalten, werden sowohl die Ausgaben des Programms als auch die Eingaben des Benutzers in eine Datei geschrieben. Der Dateiname ist im Texteingabefeld einzutragen, wobei „prt:", also der Drucker, als Defaultwert vorgegeben wird.

2.5.4.5 Aufruf eines externen Assemblers

Das integrierte „Make" kann beliebige externe Compiler und Assembler aufrufen. Wenn eine Quelldatei auf „.asm" endet und die Funktion „Text und Objekt aufnehmen" ausgewählt wird, ist es sogar schlau genug, einen Assembler-Aufruf dafür von selbst zu definieren.

Nun weiß „MaxonC++" natürlich nicht, welchen Assembler Sie verwenden (hoffentlich doch den MaxonASM!), über welchen Pfad er erreicht wird und welche Optionen nötig sind. Deshalb können Sie mit dem Symbol „Externer Assembler" im System-Dialogfenster einen Assembler-Aufruf vorgeben, den das Make defaultmäßig benutzen soll, z. B. „ma [ ] -co" oder „as -c [ ] -w26731 -lj" oder was auch immer. An der Stelle, wo das „[ ]" steht, setzt „MaxonC++" jeweils den Namen der Quelldatei ein.

2.5.4.6 Argumente aus der imaginären Kommandozeile

Beim Start von der Shell kann man Programmen Argumente übergeben, die in C mit den Parametern „argc" und „argv" ausgewertet werden können. Auch beim Programmstart aus der Entwicklungsumgebung von MaxonC++ kann man diese Art der Parameterübergabe simulieren, indem man sie im Texteingabefeld „Argumente" einschaltet. Diese Argumentzeile wird dann an aus „MaxonC++" gestartete Programme übergeben.

2.5.4.7 Compiler- und Laufzeit-Fenster

Bekanntlich öffnet „MaxonC++" für den Compiler sowie für die Programmausführung jeweils ein Fenster, das normalerweise so groß wie der Screen ist. Nun kann man aber seinen Workbench-Screen fast beliebig groß machen, und andererseits wird es auch Leute geben, die gern ein wesentlich kleineres Fenster an einer ganz bestimmten Bildschirmstelle hätten. Deshalb kann man Position und Größe dieser Fenster mit vier kleinen Symbolen im „System"-Dialogfenster festlegen.

Die beiden mit „X" und „Y" beschrifteten Symbole legen naheliegenderweise die Bildschirmposition der linken, oberen Windowecke fest, während Sie in den beiden anderen Symbole die gewünschte Breite und Höhe eintragen können. Als Default wird hier die Bildschirmgröße angenommen.

2.5.4.8 Stackgröße

Wenn ein Programm aus „MaxonC++“ gestartet wird, braucht es natürlich auch einen Stack, und im Zweifelsfall nicht zu wenig davon. Die gewünschte Größe kann ebenfalls im „System“-Dialogfenster eingestellt werden.

Nach Möglichkeit nutzt „MaxonC++“ dafür den freien Platz im Compiler-Workspace aus. Wenn dort nicht mehr genug frei ist, wird eben versucht, den Speicher anzufordern, und wenn auch das fehlschlägt, können Sie das Programm eben nicht starten.

2.5.5 Aktuelles Verzeichnis

Es ist manchmal sinnvoll, wie mit dem „cd“-Befahl das aktuelle Verzeichnis für „MaxonC++“ zu wechseln. Insbesondere ist dieses Verzeichnis wichtig, wenn in einem Quelltext Include-Dateien „so“ in normale Anführungszeichen eingebunden werden, oder wenn ein aus der Umgebung gestartetes Programm auf irgendwelche Dateien zugreift, die es nur bei passendem Verzeichnis findet.

Dafür dient der Menüpunkt „Verzeichnis“ im „Optionen“-Menü. Beachten Sie, daß „MaxonC++“ und der Editor zwei getrennte Prozesse sind und der Verzeichniswechsel sich ausschließlich auf „MaxonC++“ bezieht, und daß „MaxonC++“ im Make-Modus automatisch in das Verzeichnis wechselt, in dem die Projektdatei liegt. In diesem Fall akzeptiert „MaxonC++“ nicht so ohne weiteres den Verzeichniswechsel, sondern fragt Sie mit einem Dialogfenster, ob Sie den Make-Modus verlassen wollen.

2.5.6 Voreinstellungen mit der „Config“-Datei

Natürlich wollen Sie nicht immer, wenn Sie „MaxonC++“ starten, Ihre Lieblingseinstellungen von Hand vornehmen. Deshalb gibt es eine Konfigurationsdatei, die normalerweise im „S:“-Verzeichnis liegt und „MaxonC++.config“ heißt. Es handelt sich hier im eine ASCII-Datei, die Sie mit jedem Editor bearbeiten können.

Jeder Eintrag besteht aus einer Zeile der Form „x = y“. Sie können Kommentare setzen, die ausnahmsweise nicht C++-mäßig mit „//“, sondern mit „;“ eingeleitet werden. Die Groß-/Kleinschreibung spielt hier keine Rolle.

Zunächst einmal gibt es einige Einträge für die diversen Switches, z. B. „Autosave“. Wenn Sie diesem Symbol in der Config-Datei eine „1“ zuweisen, also

```
Autosave = 1
```

ist die Autosave-Option von „MaxonC++“ und damit auch das entsprechende Icon im System-Dialogfenster beim Programmstart automatisch eingeschaltet, während eine Null

```
Autosave = 0
```

die Option ausschaltet. Defaultmäßig, also wenn es keine Config-Datei gibt oder selbige keinen „Autosave“-Eintrag besitzt, ist diese Option übrigens eingeschaltet.

Es gibt noch weitere Switches, die alle genau wie „Autosave“ funktionieren:

„Icon“ entspricht dem gleichnamigen Symbol im System-Dialogfenster und ist per Default beim Start von der Workbench ein- und beim Aufruf aus der Shell ausgeschaltet. Acht weitere Optionen entsprechen den gleichnamigen Switches im Compiler-Dialogfenster:

Option	Symbol-Beschriftung	Defaultwert
Mode	C++	1
Break	Unterbrechen	0
Mult	IntMult 32	1
MDB	Debug-Datei	0
Symb	Symbol-Hunk	0
Asm	Asm schreiben	0
AsmSrc	Source in Asm	0
Opt	Optimieren	1
AutoExe	Exe-Datei (im Linker-Dialog)	0
AutoObj	Objektdatei (im Linker-Dialog)	0

Da wir gerade bei Compiler-Optionen sind: Auch der Compiler-Workspace läßt sich natürlich einstellen, und zwar so:

```
Work = 80
```

80 kByte ist hier auch der Default-Wert.

Der Include-Pfad bzw. die beiden Include-Pfade legt man fest mit

```
Incl = "CPP:Include"
```

bzw.

```
Incl = "ram:CPPinc", "CPP:include"
```

Fehlt noch die Einstellung für die optionalen Warnings. Hier gibt es praktischerweise nicht für jeden Switch eine eigene Option, sondern eine einzelne Zuweisung der Form

```
Warn = "revpt",
```

was übrigens auch die Default-Einstellung darstellt. Jeder Buchstabe steht für eine Warnung, die man eingeschaltet haben möchte. Es gibt folgende leicht einprägsame Kennbuchstaben - übrigens die gleichen Bezeichnungen wie bei der „-w"-Option des CPPC:

Abkürzung für		Symbol-Beschriftung
a	Assignment	Verdächtiges "="
c	Conversion	Unsichere Konv.
e	Empty Statement	Leeranweisung
f	Function prototypes	Prototypen
n	Nested comments	Kommentare
o	Old-Style	K&R-Parameter
p	Pragma	#pragma
r	Return	"return" fehlt
t	Temporary	Temporarys
v	Variable	Var. unbenutzt

Man kann auch zwei Linker-Optionen einstellen, nämlich das Datenmodell und den Lib-Pfad. Ersteres geschieht mit der Option „Data", die auf „L" (für „Large") oder „S" (für „Small") gestezt werden kann, z. B.:

```
Data = S
```

Die Festlegung des Library-Pfads (oder der beiden Library-Pfade) geht syntaktisch genau wie die Include-Pfade:

```
Libs = "CPP:Lib"
```

Ferner wären da noch diverse Systemeinstellungen, etwa die Einstellung der Sprache:

```
Lang = D
```

stellt „Deutsch" ein,

```
Lang = E
```

```
"Englisch".
```

Bei der Protokolldatei gilt es sowohl einen Ein-/Ausschalter als auch einen Dateinamen einzustellen, und zwar mit der Option „Prot“:

```
Prot = 0,"prt:"
```

„prt:“ ist übtigens auch der Default, wenn man hier keinen Dateinamen angibt.

Die Option „CallAsm“ legt den externen Assembler-Aufruf fest, den das „Make“ bei der Funktion „Text und Objekt aufnehmen“ für Assembler-Dateien vorgeben soll, und entspricht dem Eintrag im Symbol „Externer Assembler“ im System-Dialogfenster.

Auch für die simulierten Kommandozeilen-Argumente läßt sich mit der Option „Arg“ ein String vorgeben:

```
Arg = "Testargument!"
```

Da wird es Sie nicht weiter verwundern, wenn ich Ihnen sage, daß auch Position und Größe des Compiler- sowie des Laufzeitfensters festgelegt werden können, und zwar in der kanonischen Reihenfolge X,Y,B,H:

```
Window = 0,10,0,0
```

Obige Einstellung öffnet für Compiler und laufendes Programm jeweils ein Fenster, daß am oberen Rand zehn Punkte freiläßt, sonst aber den gesamten Screen füllt.

Der Assembler-Aufruf, den die Funktion „Text und Objekt aufnehmen“ defaultmäßig verwenden soll, wird mit der Option „CallAsm“ festgelegt, z. B. so:

```
CallAsm = "asm [ ]"
```

Was fehlt noch? Eine Option für den Laufzeitstack. Hier ist sie:

```
Stack = 20
```

Natürlich erfolgt die Angabe in kByte.

Es gibt in der Config-Datei aber auch eine Einstellung, für die es in „MaxonC++“ kein Symbol gibt: Den Namen des Editors. Schließlich muß „MaxonC++“ den Edward nachladen und dafür Pfad und Dateinamen kennen. Die Option heißt naheliegenderweise „Ed“ und ist mit folgendem Default-Wert belegt:

```
Ed = "Edward:Edward"
```

3. MAKE - eine integrierte Projektverwaltung

3.1 Das Prinzip

3.1.1 Wozu eine Projektverwaltung?

Größere Projekte bestehen normalerweise nicht aus einer einzelnen Quelltextdatei, sondern aus einer Vielzahl von Modulen, die vom Linker verbunden werden - zumindestens sollte es so sein. Das nimmt leicht Ausmaße an, die nicht mehr ganz leicht zu überschauen sind. Hinzu kommt, daß es natürlich Unsinn ist, nach jeder Änderung das ganze Projekt zu übersetzen. Auf die Dauer wird es aber lästig, sich immer wieder neu zu überlegen, welches Modul neu übersetzt werden muß, einmal ganz davon abgesehen, daß man eigenartige Überraschungen erleben kann, wenn man dabei etwas vergißt. So entsteht das Bedürfnis, solche Vorgänge zu automatisieren, wozu hat man schließlich einen Computer!

Die populärste Projektverwaltung, die auf so ziemlich allen ernstzunehmenden Rechnern anzutreffen und bei allen Compilersprachen tausendfach bewährt ist, heißt „make". Eigentlich ist „make" selbst eine Art Programmiersprache, und das macht es natürlich nicht leicht, ein komplettes Make in eine graphisch orientierte Entwicklungsumgebung wie die von MaxonC++ zu integrieren. Deshalb unterstützt MaxonC++ nur einen recht begrenzten aber den entscheidenden Teil von „make".

3.1.2 Was ist das überhaupt - ein Projekt?

Fangen wir doch einmal ganz graswurzelmäßig von vorn an: Woraus besteht ein Projekt (immerhin war von einer Projekt-Verwaltung die Rede), und was ist überhaupt ein Projekt?

Also, zu den oben erwähnten Einschränkungen gehört, daß ein Projekt immer genau ein ausführbares Programm darstellt. Wenn man im Rahmen eines Projekts ein Programm in mehreren Versionen erstellen will (man denke da nur an „MaxonC++" und „CPPC", die im wesentlichen ein und dasselbe Programm sind, nur mit unterschiedlichen aufgepfropften Benutzerschnittstellen), muß man eben für jedes einzelne Executable eine eigene Projektdatei erstellen.

OK - ein Projekt ist also ein ausführbares Programm. Und wie kommt man zu einem solchen Programm, sofern es sich nicht gerade um ein Bonsai-Miniatur-Projekt handelt? Richtig, das erwähnte ich schon: Man linkt dafür Objektdateien, also Module, zusammen. Genaugenommen bindet der Linker auch noch fertige Bibliotheksmodule ein, aber

das interessiert uns hier nur höchst periphär. Der Freizeit-Existenzialist stellt sich jedenfalls nun die Frage „Und wo kommen die kleinen Module her?", der Coder weiß es schon: Das hat ausnahmsweise nichts mit Schweinskrams zu tun, sondern die Objektdateien werden von irgendwelchen Compilern erzeugt.

Zu jedem der Module gibt es genau einen ganz besonderen Quelltext, nämlich den Text, aus dem die Objektdatei durch Compilieren entsteht. Um so ein Objekt zu machen, braucht die Projektverwaltung also im wesentlichen eine Information darüber, wie der Quelltext dazu heißt, und außerdem noch ein paar Zusatzinformationen, etwa den Namen des Compilers oder irgendwelche der allseits beliebten Compiler-Optionen.

Aber das ist noch lange nicht alles, denn zu einen normalen Projekt gehören noch einige Texte mehr, normalerweise Includedateien, die von den eigentlichen Quelltexten irgendwo und irgendwie benutzt werden. Das führt uns natürlich zu der Frage, wann und wie eine Objektdatei neu übersetzt wird. Nun, bevor wir alle in Rente gehen, werden wir vielleicht noch das Entwicklungssystem erleben, das Programme on-line compiliert, während man den Quelltext bearbeitet - so, wie es jetzt schon der Kick-Ass tut, etwas Schleichwerbung in eigener Sache muß schließlich erlaubt sein. Vorerst ist es jedenfalls noch so, daß man munter an seinen Quelltexten ‚rumediert und irgendwann entscheidet, daß das Programm für eine erneute Übersetzung reif ist. Dann tritt die bereits mehrfach erwähnte Projektverwaltung „Make" in Aktion und schaut nach, welche Objektdateien für das Projekt nötig sind, und bevor diese zusammengelinkt werden, denkt es scharf darüber nach, welche der Objektdateien neu übersetzt („recompiliert") werden müssen. Da ja der Compiler wenigstens zeitweise mit dem Programmierer Schritt halten soll, ist auch noch halbwegs einsichtig, wann das zu geschehen hat: Richtig genau dann, wenn mindestens einer der Texte, aus denen das Modul entsteht (d. h. Quelltext und Include-Files), seit der letzten Übersetzung verändert wurde. Das läßt sich notfalls leicht an Erstellungsdatum und -uhrzeit der entsprechenden Dateien feststellen.

3.1.3 Was hat das ganze mit „Make" zu tun?

Die Projektverwaltung verwaltet im wesentlichen zweierlei Datenbestände: Texte und Objekte. Bei den Texten handelt es sich im allgemeinen um irgendwelche Quelltexte oder Includedateien, und aus diesen Texten erzeugt der Compiler unter der Kontrolle der Projektverwaltung die Objektdateien. Im letzten Schritt werden diese Objektdateien dann mit den Libraries zu einer ausführbaren Datei gelinkt.

Für jede der Objektdateien braucht das „Make" also Angaben darüber, wie es „gemacht" (daher der Name) werden soll, insbesondere den Namen eines Quelltexts und eines Compilers. Das würde man natürlich auch noch von Hand hinkriegen, und deshalb sagt man der Projektverwaltung auch, unter welchen Umständen ein Objekt compiliert werden soll und wann die alte Version noch einmal verwendet werden kann.

Ein Objekt muß neu übersetzt werden, wenn ein Text, der mit diesem Objekt zu tun hat, verändert wurde. Normalerweise handelt es sich bei diesen Texten um den Quelltext des Moduls sowie die darin benutzten Includedateien. Dazu vergleicht „Make" den Datestamp, also der Zeit der letzten Veränderung einer Datei, der Objektdatei mit denen „ihrer" Texte. Natürlich wird ein Objekt auch dann compiliert, wenn es bisher noch nicht existierte.

3.2 Die Praxis

3.2.1 Erstellen von Projekten

Alle Angaben über die Struktur eines Projekts werden sinnvollerweise in einer Datei festgehalten, dem Makefile. Mit MaxonC++ erstellen Sie diese Datei interaktiv mit dem Pull-Down-Menü „Make". Das Makefile wird dabei ständig aktualisiert, also nach jeder Veränderung neu abgespeichert. Da eine solche Datei normalerweise nicht sehr lang sein wird, dürfte dabei auch beim Diskettenbetrieb keine Verzögerung auftreten.

Am besten probieren Sie das Make-Utility einfach aus, indem Sie sich ein Mini-Projekt aus zwei Modülchen erstellen. Damit fangen Sie an, indem Sie den Menüpunkt „Neues Makefile" anwählen. Nun erscheint das Dateiauswahlfenster und fragt Sie nach dem gewünschten Dateinamen. Meist nennt man die Datei ganz einfach „makefile", jedenfalls dann, wenn man nicht mehr als eine solche Datei in einem Verzeichnis hat. Wenn Sie dagegen an einem vorhandendem Projekt weiterarbeiten wollen, benutzen Sie den Menüpunkt „Makefile laden".

MaxonC++ wechselt immer automatisch in das Verzeichnis, in dem die aktuelle Makedatei liegt. Um Konfusion zu vermeiden, wird die Funktion „Verzeichnis" im Menü „Projekt" gesperrt, wenn ein Makefile geladen oder erzeugt wird.

3.2.2 Hinzufügen von Dateien, umständliche Methode

Zu einem Projekt gehört natürlich mindestens eine Objektdatei, und um eine solche zu erzeugen, braucht man einen Quelltext. Versuchen wir es zunächst auf die umständliche Methode und schreiben einen Text, z. B.

```
#include <stream.h>

void test()
{ cout « "Dies ist ein Test."; }
```

Diesen Text speichern Sie einmal als „test.c“ ab (mit einer Datei ohne Namen kann „Make“ nichts anfangen) und wählen dann den Menüpunkt „Make/Text aus Editor hinzufügen“. Nun wird die Datei dem Projekt hinzugefügt, und „Make“ weiß jetzt, daß „test.c“ irgendwie dazugehört. Sie können auch den Manüpunkt „Text aus Datei hinzufügen“ benutzen, dann müssen Sie die Datei mit dem Dateiauswahlfenster auswählen.

Als nächstes sagen wir dem Make, daß unser Programm aus einem Objekt namens „test.o“ entstehen soll. Also wählen wir den Menüpunkt „Make/Objekt hinzufügen“ und tippen dann im Dateiauswahlfenster den Namen der bisher nicht existierenden Datei „test.o“ ein, vorzugsweise im selben Verzeichnis wie „test.c“ und das Makefile. Und dumm, wie Computer nun einmal sind, müssen wir ihm jetzt auch noch klarmachen, daß „test.o“ entsteht, indem man „test.c“ compiliert.

Dazu dient der Menüpunkt „Compileranweisungen“. Im Dialogfenster, das daraufhin erscheint, wählt man den Namen der Objektdatei aus. Dann erscheint ein weiteres Dialogfenster, in dem man den Compiler-Modus (C++, ANSI C und „unbekannt“) und die Quelltextdatei auswählen kann. Klicken Sie hier einfach „test.c“ als Source an und stellen Sie den Compiler-Schalter auf „C++“.

Wenn Sie jetzt das Programm compilieren (Taste F9 oder Menü „Compiler/Übersetzen“), tritt Make in Aktion, stellt fest, daß „test.o“ compiliert werden muß und bindet anschließend aus diesem Objectfile ein Programm zusammen oder versucht es zumindest, denn es gibt ja noch keine „main“-Funktion. „Make“ funktioniert völlig unabhängig davon, welcher Text sich gerade im Editor befindet.

Wenn „Make“ ein Objekt erzeugt, benutzt es dazu den Compilermodus, der im „Compileranweisungen“-Dialogfenster eingestellt wurde, also ANSI C oder C++. Wenn man hier „None“ einstellt, gibt „Make“ lediglich eine Meldung aus, daß dieses Modul neu zu übersetzen ist.

Dieser Modus wird vor allem dann gebraucht, wenn man in sein Projekt Objektdateien anderer Compiler oder Assembler einbinden will, auch wenn man in diesem Fall meist die Kommandozeilenversion des Compilers vorziehen wird.

Das war jetzt wie gesagt die umständliche Methode. Beim folgenden Miniatur-Modul, das die noch fehlende „main“-Funktion enthält, machen wir es etwas einfacher:

```
#include "test.h"

void main()
{
  test();
}
```

Diesen Text speichern wir einfach als „main.c" ab und wählen den Menüpunkt „Tx.+Obj. hinzufügen". Jetzt macht Make vollautomatisch folgendes:

- Die Datei „main.c" wird in die Liste der Texte aufgenommen.
- Das Objekt „main.o" wird dem Projekt hinzugefügt.
- Eine entsprechende Compileranweisung wird gesetzt, und zwar mit dem Modus, der gerade im Menü „Optionen/Compiler/C++" eingestellt ist.

Falls der Name der Quelldatei jedoch auf „.asm" endet, wird als Modus „Extern" gewählt und ein externer Assembleraufruf entsprechend der Einstellung im System-Dialogfenster vorgegeben. Ferner wird eine Abhängigkeit der Objektdatei von der Quelldatei eingetragen, aber diese Abhängigkeiten werden erst weiter unten erläutert.

Jetzt fehlt dem Projekt nur noch eins: Die Header-Datei „test.h", in der die Funktionen des Moduls „test" deklariert. Die Datei enthält lediglich folgendes:

```
void test();
```

also den Prototypen der Funktion. Wenn Sie jetzt den Compiler starten, müßte das Programm korrekt übersetzt und gelinkt werden.

Das so ziemlich wichtigste Feature der Projektverwaltung haben wir aber bisher nicht benutzt: die Abhängigkeiten. Offensichtlich benutzt „main.o" die Datei „test.h" als Includedatei und ist folglich davon abhängig. Immer, wenn „test.h" verändert wurde, sollte deshalb auch „main.o" sicherheitshalber neu übersetzt werden.

Wählen Sie dazu den Menüpunkt „Abhängigkeiten" und anschließend das fragliche Object-File aus dem daraufhin erscheinenden Dialogfenster aus. Im zweiten Dialogfenster, das sich nun öffnet, klicken Sie das Symbol „test.h" an, und schon weiß „Make" Bescheid.

3.2.3 Weitere Funktionen

Natürlich kommt eines Tages auch der Zeitpunkt, an dem ein Text oder ein Objekt aus einem Projekt ,rausfliegt. Dazu dienen die beiden Menüpunkte „Objekt löschen" und „Text löschen". Letzterer entfernt den Text auch aus sämtlichen Compileranweisungen und Abhängigkeiten, in denen er vorkommt.

Ausgesprochen praktisch ist „Text laden", denn damit kann man sehr bequem einen Text des Projekts in den Editor laden. Dabei wird die Funktion-„Öffnen"des Edward benutzt.

„Objekt markieren" ist in einem gut gepflegten Makefile eigentlich überflüssig, aber wer ist schon perfekt... Wenn Sie in diesem Menü eine Objektdatei anwählen, wird diese beim nächsten Compilieren auf jeden Fall neu übersetzt. Von schlampig definierten Abhängigkeiten einmal abgesehen, braucht man dieses Feature auch, wenn man irgendwelche Compileroptionen verändert (z. B. das ganze Projekt von „Small" auf „Large Data" umgestellt oder die Optimier-Option erstmals eingeschaltet) hat.

Und last not least kann man natürlich auch den Make-Modus wieder verlassen und „normal" weiterarbeiten. Dazu dient natürlich der Menüpunkt „Make verlassen".

3.2.4 Externe Übersetzungen

Ein Projekt kann natürlich auch Module enthalten, die nicht in C oder C++ geschrieben sind oder aus anderen Gründen nicht mit MaxonC++, sondern mit anderen Compilern oder Assemblern übersetzt werden müssen. Außerdem können Sie auch Objektdateien haben, zu denen Sie den Quelltext nicht besitzen, z. B. weil jener Teil des Projekts von einem Kollegen geschrieben wird.

Betrachten wir doch zuerst den Fall, daß Sie eine Objektdatei in Ihrem Projekt haben, die aus einem der genannten Gründe nicht vom MaxonC++ erzeugt werden kann. Den Fall können Sie mit dem in 3.2.2 geschilderten leicht lösen: Sie fügen das Modul mit „Objektdatei aufnehmen" in Ihr Make-File ein, und das war's dann.

Halt, das war's noch lange nicht! MaxonC++ hat auch hier noch ein paar Tricks auf Lager, die das Leben des Programmierers (wieder einmal) etwas angenehmer machen. Sehen Sie sich doch einmal den „Compileranweisungen"-Dialogfenster zu so einer quelltextlosen Datei an: Als Compilermodus ist hier „None" angegeben. Trotzdem können Sie aber einen Quelltext angeben. Ein Quelltext zu einer Objektdatei, die nicht übersetzt wird (jedenfalls nicht vom Maxon Make) ist gleichbedeutend mit einer Abhängigkeit, denn auch diese kann man für so eine Objektdatei setzen. Die Projektverwaltung prüft dann, ob die Objektdatei existiert und ob sie vielleicht älter als ihr Quelltext oder als eine ihrer Abhängigkeits-Dateien ist. Falls ja, wird eine Meldung der Art „Datei xyz.o muß extern recompiliert werden" ausgegeben. Damit wissen Sie, was Sie zu tun haben.

Wenn Sie schon ein wenig mit den Compileranweisungen 'rumgespielt haben, wird Ihnen aufgefallen sein, daß es außer „ANSI C", „C++" und „None" noch einen vierten Modus gibt, nämlich „Extern". Dabei ist dann das bisher gesperrte Texteingabefeld „Anweisung" wählbar, während die Quelltext-Symbolleiste komplett deaktiviert ist. In diesem Modus können Sie eine beliebige CLI-Anweisung (z. B. einen Aufruf eines externen Assemblers) eingeben, nach Möglichkeit natürlich eine solche, die die Objektdatei wirklich erzeugt (auch wenn man hier natürlich auch 'echo „Hello, World!"'

eingeben kann). Das Make öffnet bei der Übersetzung dann ein zusätzliches Textwindow und führt die Anweisung darin aus. Achten Sie aber bitte darauf, daß die externe Anweisung eine Objektdatei und keine ausführbare Programmdatei erzeugt, denn damit kann der Linker nichts anfangen.

Außerdem sollte die Projektverwaltung wissen, wann sie diese externe Übersetzung ausführen lassen soll. Deshalb sollten Sie unbedingt den Quelltext dieses Moduls sowie alle seine Includedateien in das Projekt aufnehmen und die Abhängigkeiten korrekt setzen.

Während Mischungen von C++ mit Pascal, ADA, Occam oder anderen Hochsprachen in der Praxis nicht so häufig vorkommen, muß man doch des öfteren ein Assembler-Modul in sein Programm einfügen. Wenn Sie einen Assembler besitzen, der sich als CLI-Kommando aufrufen läßt, z. B. die entsprechende Version des Maxon Assemblers, liegt es natürlich nahe, den Assembleraufruf auf diese Art und Weise in die Make-Datei aufzunehmen. Und weil das oft genug vorkommt, bietet MaxonC++ Ihnen auch hier ein zusätzliches Stück Komfort: Wenn Sie im Editor einen Text haben, dessen Name auf „.asm" endet - also offensichtlich ein Assembler-Quelltext - und dann die Funktion „Text und Objekt aufnehmen" anwählen, nimmt „MaxonC++" die „.asm"-Quelldatei sowie ihre „.o"-Objektdatei auf und setzt dafür sogar schon den richtigen Assembler-Aufruf. Ferner legt es eine Abhängigkeit der Objektdatei von der Quelldatei fest, ganz wie sich das gehört.

Nun weiß „MaxonC++" natürlich nicht, welchen Assembler Sie verwenden und welche Optionen Sie bevorzugen. Deshalb gibt es im „System"-Dialogfenster ein Texteingabefeld mit der Beschriftung „Externer Assembler", in das man eine Rohform des Assembleraufrufs eintragen kann, z. B. „masm -co [ ]". „MaxonC++" übernimmt bei „Text und Objekt aufnehmen" diesen Aufruf und fügt anstelle des „[ ]" den Quelltextnamen ein. Damit dürfte es kein Problem mehr darstellen, Ihren Lieblingsassembler unter „MaxonC++" aufrufen zu lassen.

3.3 Handgemachte Makedateien

Das Format der Projektdateien von MaxonC++ orientiert sich an den Standard-Makefiles, ist aber natürlich etwas strikter, da in der integrierten Entwicklungsumgebung ja nur ein gewisser Ausschnitt des bekannten Make implementiert werden konnte. Es ist dadurch relativ leicht, ein solches Makefile von Hand zu erstellen oder ein vorhandenes Makefile an MaxonC++ anzupassen. Wenn Sie nicht in der integrierten „MaxonC++"-Umgebung arbeiten, sondern die Option „-m" des CPPC benutzen, bleibt Ihnen sogar nichts anderes übrig, als Make-Dateien grundsätzlich manuell zu erstellen.

Es gibt zwei verschiedene Arten von Zeilen, nämlich Abhängigkeits- und Anweisungszzeilen. Sie unterschieden sich formal dadurch, daß letztere mit einem Leerzeichen beginnen. Auf jede Abhängigkeitszeile kann optional eine Anweisungszeile folgen, aber nicht mehr als eine.

Die erste Abhängigkeitszeile muß immer mit dem Namen der Exe-Datei beginnen, in unserem Beispiel etwa „test". Es ist nicht nötig, weitere Abhängigkeiten in diese Zeile zu schreiben. Normalerweise würde man hier die Namen der Objektdateien angeben, aber MaxonC++ ist eben nicht normal.

Die zugehörige Anweisungszeile muß mit der Kennung „CPPC" beginnen, gefolgt von den Objekten des Projekts. Für unser Beispiel fängt die Makedatei also so an:

```
Test:
  CPPC main.o test.o
```

Nun folgen die Abhängigkeits- und Anweisungszeilen der übrigen Objektdateien:

```
test.o: test.c
  C++ test.c
main.o: main.c test.h
  C++ main.c
```

Die Anweisungszeile beginnt mit „C++", wenn das Modul im C++-Modus übersetzt werden soll, oder mit „cc" für ANSI C-Programme. Bei unbekanntem Modus („None") ist hier das vielsagende „from" zu wählen, sofern man überhaupt das Bedürfnis hat, dafür eine Compileranweisung anzugeben (das ist dann eigentlich nur eine weitere Abhängigkeits-Deklaration). Jede andere Compileranweisung wird als externer Aufruf interpretiert.

Es ist übrigens nicht zwingend erforderlich, dem Make zu sagen, daß ein Objekt von seiner Quelltextdatei abhängig ist, denn das weiß es auch so. Also könnte man das Makefile folgendermaßen abkürzen:

```
Test:
  CPPC main.o test.o
test.o:
  C++ test.c
main.o: test.h
  C++ main.c
```

Aber auch das ist noch nicht optimal: Wenn Sie hinter den Namen des Exe-Files einige „.o"-Dateien angeben, geht Make davon aus, daß diese Objekte, sofern nicht anders

angegeben, aus einer entsprechenden „.c"-Datei entstehen, wobei es den Compilermodus wählt, der gerade als Default im „Optionen"-Menü bzw. über die Kommandozeilenoptionen „-pc"/"-pp" eingestellt ist. Mit diesem Feature läßt sich die Makedatei noch weiter abkürzen:

```
Test: main.o test.o
CPPC main.o test.o
main.o: test.h
```

Ein Projekt besteht aber oft nicht nur aus Objektdateien und Standard-Bibliotheken, sondern benötigen auch Linker-Bibliotheken wie die allseits beliebte „amiga.lib". In „MaxonC++" werden diese bekanntlich im entsprechenden Symbol des Linker-Dialogfensters eingetragen, wobei sie dann gegebenenfalls automatisch in das Makefile aufgenommen werden. Falls Sie die Option „-m" des CPPC benutzen, können Sie solche Angaben wie alle anderen Optionen auch in die Kommandozeile aufnehmen, etwa

```
CPPC -w80 -gO -m makefile -l amiga
```

oder wie auch immer. Sie können die Libraries aber auch direkt in das Makefile eintragen, und zwar als Option hinter der ersten Anweisungszeile (der mit dem „CPPC"):

```
Test: main.o test.o -l amiga
CPPC main.o test.o
main.o: test.h
```

Mit diesen Informationen sollte es für Sie kein Problem mehr darstellen, eine MaxonC++-konforme Projektdatei von Hand zu erstellen.

4. Der Linker

4.1 Prinzipielles

Ein Linker erfüllt im wesentlichen zwei Aufgaben:

- Er fügt Module eines Programms zusammen. Das ist ein nahezu unverzichtbares Feature, denn ab einer gewissen Größe ist es praktisch unmöglich, ein Programm in einem einzigen Modul zu schreiben.

- Anschließend bindet er vordefinierte Bibliotheksroutinen in das Programm ein. Auch das ist eine Aufgabe, die aus dem Programmieralltag nicht wegzudenken ist, denn jeder C- oder C++-Programmierer erwartet ganz selbstverständlich, daß simple Funktionen wie „strcpy" oder aufwendige Operationen wie „printf" in irgendwelchen Bibliotheken vorhanden sind und vom Linker anstandslos eingebunden werden.

Diese beiden Funktionen überschneiden sich allerdings, denn der Unterschied besteht einzig und allein darin, daß man dem Linker ausdrücklich sagt, welche Module er in ein Programm einbinden soll, während er die benutzten Bibliotheksfunktionen weitgehend selbständig hinzulinkt - und, wenn er hinreichend dumm ist, auch noch erheblich mehr - vor allem im Workstation-Bereich gibt es Compiler-/Linker-Systeme, bei denen ein „Hello, World!"-Programm reichlich 50 KByte groß ist, weil der Linker gnadenlos riesige Bibliotheken komplett einbindet.

Das ist es, was man allgemein von einem Linker erwartet, zumal die heute gebräuchlichen Linker immer noch auf Konzepten der 50er Jahre basieren, wie Stroustrup einmal sagte. Auch der Standard-Linker des Amiga, ALink die verbesserte Version, „BLink" oder die diversen Nachbauten dieser Linker tun im großen und ganzen nichts anderes. Auch der integrierte Linker von MaxonC++, der zu Link/BLink kompatibel ist, beherrscht diese beiden Aufgaben, wenn auch zum Teil etwas eigenwillig.
Nun reden wir hier aber von C++ und nicht von solchen Antiquitäten wie Fortran oder C, und die neue Zeit fordert nicht nur neue Compiler, sondern stellt auch neue Anforderungen an die damit verbundenen Linker.

Eine der zahlreichen, auf den ersten Blick eher unscheinbaren Neuerungen, die C++ gegenüber C so auf dem Kasten hat, ist die Initialisierung statischer Daten. In C dürfen globale Variablen ausschließlich mit konstanten Werten initialisiert werden, weshalb eine Deklaration wie

```
int i = 42;
```

weder Compiler noch Linker vor unlösbare Probleme stellt: Die Variable „i" wird im Datenhunk des Programms angelegt, und die entsprechende Speicherstelle bekommt von Anfang an den Wert „42". Schon in dem Augenblick, wenn das Programm in den Speicher geladen wird, steht also an der Stelle, an der sich später einmal die Variable „i" befinden wird, das entsprechende Langwort, und das Schönste daran ist, daß das den Linker überhaupt nicht zu interessieren braucht, denn er bekommt auf diese Weise gar nicht mit, daß es eine solche Initialisierung gibt.

In C++ ist die ganze Angelegenheit schon etwas fortschrittlicher, denn dem Programmierer werden in seinen Initialisierungsdrang keinerlei Zügel mehr angelegt. Das folgende ist zum Beispiel ein korrektes C++-Programm:

```
#include <stdio.h>

int murx = printf("Sachen gibt\'s!!!");

void main() { }
```

Obwohl das Hauptprogramm „main" scheinbar leer ist, macht dieses Programm etwas, denn noch vor der Ausführung von „main" wird die globale Variable „murx" mit dem Ergebnis eines „printf"-Aufrufs initailisiert (das Ergebnis von „printf" ist die Anzahl der ausgegebenen Zeichen oder ein Fehlercode). Also passiert hier schon eine Ausgabe, bevor überhaut das scheinbare Hauptprogramm gestarten wird.

In dieser Form ist das natürlich ein eher krankhafter Fall von Programmverschleierung, aber durch Konstruktoren kommt es in C++ oft zu solchen absolut unkonstanten Initialisierungen. Die ganz besondere Pointe ist, daß jedes Modul solche Initialisierungen besitzen kann. Lange Rede, schwacher Sinn: Der Linker muß in C++ nicht bloß linken, sondern auch dafür sorgen, daß Module korrekt initialisiert werden - ja, nicht nur das: falls globale Variablen einen Destruktor haben, muß er sogar dafür sorgen, daß nach Programmende derartige Destruktoren aufgerufen werden.

Sie werden lachen, aber der Maxon-Linker kann das! Wie das genau vor sich geht, wird an anderer Stelle erläutert. Kommen wir statt dessem noch einmal auf die beiden anfangs erwähnten Aufgaben des Linkers zu sprechen.Wie Sie MaxonC++ dazu bringen, Module zusammenzubinden, dürfte aus den vorhergehenden Kapiteln hoffentlich klargeworden sein (Nämlich in der integrierten Umgebung mit einem Makefile und beim „CPPC" wahlweise mit einem Makefile oder entsprechenden Kommandozeilen-Argumenten). Deshalb widmen die beiden folgenden Abschnitte sich der anderen Funktion, dem Einbinden von Bibliotheksfunktionen:

4.2 Standard-Funktionen und das „Logfile“

Der Linker von MaxonC++ beherrscht das Einbinden von Bibliotheksroutinen erstaunlicherweise gleich auf zwei verschiedene Arten. In diesem Abschnitt soll von der etwas unkonventionelleren Methode die Rede sein, nämlich dem Linken mit Objekt-Dateien über eine Datei namens „Logfile“.

Im großen und ganzen ist eine Bibliothek nichts anderes als eine Sammlung von Modulen mit nützlichen Funktionen. Der Linker geht nun hin und nimmt sich selbsttätig aus dieser Sammlung alle Module, die gerade benötigt werden, und benötigt werden eben jene Objekt-Dateien, in denen Symbole (also z. B. Funktionsnamen) deklariert werden, die im aktuellen Programm benutzt werden, bisher aber nicht definiert wurden.

Es ist deshalb ein naheliegender Gedanke, dem Linker in Form einer Datei mitzuteilen, welche Objektdatei er einlinken soll, wenn ein bestimmtes Symbol sonst nicht bekannt ist. Genau dieses genial einfache, aber zugegebenermassen nicht sehr kompatible Prinzip wurde bei MaxonC++ implementiert: Im „lib“-Verzeichnis, in dem sämtliche Linker-Libraries stehen, gibt es eine Datei namens „logfile“, welche Informationen darüber enthält, was wo steht. Der Dateiname wurde etwas eigenartig gewählt, denn normalerweise versteht man unter einem „log“ so etwas wie ein Protokoll - jeder TeX-User kennt die „.log“-Files. Hier ist das „log“ ganz einfach eine legasthenische Verkürzung von „Katalog“. Wenn ich ehrlich sein soll, habe ich das „logfile“ so genannt, weil das einfach gut klingt und mir gerade nichts besseres einfiel. Wesentlich interessanter dürfte für Sie sein, was in dieser Datei enthalten ist.

Das „logfile“ ist eine ganz gewöhnliche Textdatei, die man leicht mit jedem Editor bearbeiten kann, und besteht aus einer Folge von Einträgen wie

```
Dateiname1
  Symbol1
  Symbol2
  Symbol3
Dateiname2
  Symbol4
usw.
```

Eine Zeichenfolge, die in der ersten Spalte beginnt, stellt also den Namen einer Objektdatei dar, die sich nach Möglichkeit ebenfalls im „libs“-Verzeichnis befinden sollte. Falls jedoch im Dateinamen ein „:“ oder „/“ vorkommen sollte, kann die Objektdatei liegen, wo sie will, Hauptsache der Pfad wurde auch richtig angegeben.

Jedem Dateinamen kann eine nahezu beliebige Folge von Symbolnamen folgen, die sich dadurch auszuzeichnen haben, daß sie in ihrer Zeile um mindestens ein Leerzeichen eingerückt sind. In diesem Zusammenhang sei davor gewarnt, hier „TAB“s zu benutzen, denn die mag der Linker überhaupt nicht.

Zur Steigerung der Anschauung hier ein Auszug aus dem originalen Logfile:

```
startup.o
        __StartUp
dosbase.o
        exit_i
        _exit
        abort_standard
        _DosBase
        _SysBase
        _Filehandle_Std_Input
        _Filehandle_Std_Output
        _cin
        _cout
        _cerr
        _std__in
        _std__out
        _std__err
        _errno
        __Writefile
        __Writefile__NoBuf
wbstartup.o
        _wbmain
stdinput.o
        _getc
        getc__P06stream
        _fgetc
        fgetc__P06stream
        _getchar
        getchar_
        __Readfile
        __Readfile__NoBuf
arg.o
        main_
```

Es gibt hier einige (übrigens in Assembler geschriebene) Objektdateien, in denen bestimmte Linker-Symbole deklariert werden, zumindens behauptet das Logfile das. Wenn also in einem Programm ein Symbol namens „_getc“ referiert wird, weiß der Linker, daß er die Datei „stdinput.o“ aus dem „libs“-Verzeichnis einbinden soll.

Es gibt im wesentlichen für jede Funktion eine eigene Objektdatei. Oft wurden aber aus rein praktischen Erwägungen Funktionen mit gemeinsamen Unterroutinen (z. B. „getc“ und „fgetc“) oder sinngemäß zusammenhängende Funktionen (kein „fopen“ ohne „fclose“) in einer Objektdatei zusammengefaßt. Effektiv wird hier also nur das eingebunden, was auch tatsächlich benutzt wird.

Damit dürfte auch klar sein, wie Sie selbst die Standard-Bibliotheken erweitern können: Kopieren Sie einfach die Objektdatei in das „libs"-Directory, und fügen Sie dem Logfile einen entsprechenden Eintrag hinzu. Wenn Sie sich nicht sicher sind, wie die Linkersymbole zu ihren Routinen heißen - was in C++ ja nicht so ganz trivial ist, da ja die Parameterlisten in den Symbolnamen hineincodiert werden, gibt es zwei Möglichkeiten, diese herauszufinden: Entweder lassen Sie sich ein Assembler-Listing erzeugen und suchen sich dort die „XDEF"-Zeilen heraus, oder Sie versuchen zuerst einmal „einfach so", die selbstgeschriebenen Routinen einzubinden, und merken sich dann, was der Linker nicht finden kann.

4.3 Das Hauptprogramm und seine Argumente

Wenn Sie sich das Logfile durchlesen, erweckt vielleicht der Eintrag „main_" für die Datei „arg.o" Ihr berechtigtes Interesse, auch wenn dieser leider eben nicht besagt, daß man das Hauptprogramm überhaupt nicht mehr selbst schreiben muß. „main_" ist in ++-Linkage der Name des parameterlosen Hauptprogramms. Nun darf man „main „natürlich auch mit den beiden kanonischen Argumenten deklarieren, also so:

```
int main(int, char**)
```

Wird dies im C++-Modus übersetzt, lautet das zugehörige Linkersymbol „main_iPPc", und im C-Modus wird der Programmname für den Linker immer in „_main" übersetzt.

Dieses Dilemma löst MaxonC++ elegant dadurch, daß der Startup-Code zunächst in eine Funktion „main_" einspringt. Wenn der Linker dieses Symbol nicht findet, liegt das entweder daran, daß der Programmierer das Hauptprogramm schlicht vergessen hat, oder es ist eben darauf zurückzuführen, daß „main" mit Parametern oder im C-Modus deklariert wurde. Dem Linker ist dies zunächst einerlei, und da das Symbol „main_" im Startup-Code referenziert, aber nirgendwo definiert wird, sucht er es wie jedes andere Symbol in den Bibliotheken. Dort wird er auch prompt fündig und bindet die Objektdatei „arg.o„ ein. Hinter dem Symbol „main_" verbirgt sich in diesem Bibliotheksmodul eine kleine Routine, die den Argumentvektor „argv" erzeugt und initialisiert, „argc" und den Zeiger auf „argv" auf den Stack ablegt und dann in eine Routine „main_iPPc" einspringt.

Wurde das Hauptprogramm im C++-Modus mit den üblichen Parametern deklariert, trägt es eben jenes Symbol, und die Sache ist für den Linker gegessen. Im C-Modus muß er noch einen weiteren Schritt machen: er bindet zusätzlich das Modul „another_main.o" ein, in dem ein Symbol „main_iPPc" deklariert wird. In dieser Mini-Datei steht nichts anderes als ein Sprung an eine Adresse namens „_main", und auf diese Art und Weise wird auch in C-Programme korrekt eingesprungen.

Das Angenehme an dieser Rumlinkerei ist, daß auf diese Weise der Code für die Berechnung von „argv" in C++ nur dann eingebunden wird, wenn das Programm die Argumente auch auswertet, das spart einige Bytes an Codegröße. Im C-Modus wird dieser Code leider immer eingebunden, und hinzu kommen noch 8 Bytes sowie ein Relokationstabelleneintrag für den zusätzlichen Sprung.

Ach ja, noch was: Es gibt auch eine Bibliotheksfunktion „_wbmain". Wenn Sie also nicht selbst eine Startup-Routine für die Workbench schreiben, nimmt der Linker die aus der Bibliothek. Diese Funktion macht aber nichts anderes als umgehend auszusteigen. MaxonC++ geht davon aus, daß Programme, die nicht eigens dafür geschrieben sind, von der Workbench gestartet zu werden, nicht künstlich Workbench-fähig gemacht werden sollten (etwa durch zwangsweises Öffnen eines Text-Fensters oder ähnlichen Schnickschnack). Daß ein MaxonC++-Programm also nicht ohne weiteres von der workbench startbar ist, ist kein Bug, sondern ein Feature.

4.4 Linker-Bibliotheken

Ich glaube, ich erwähnte schon, daß die Sache mit dem Logfile eine Spezialität von MaxonC++ ist. Der große Vorteil ist dabei, neben der einfachen und flexiblen Erweiterbarkeit, die recht hohe Geschwindigkeit vor allem bei langsameren Speichermedien: Der Linker hat detaillierte Informationen darüber, was er einbinden soll, und muß dann nur noch die benötigten Dateien laden.

Gebräuchlicher sind hingegen Linker-Libraries, bei denen die gesamte Bibliothek in einer Datei zusammengefaßt wird. Der Linker liest sich diese Datei aufmerksam durch und pickt sich die benötigten Teile heraus. Interessant ist dieses Verfahren vor allem deshalb, weil die offiziell von Commodore vertriebene „amiga.lib" in einem solchen standardisierten Format vorliegt - natürlich alles „ALink/BLink"-kompatibel.

Deshalb unterstützt auch der MaxonC++ Linker sinnvollerweise dieses Bibliotheksformat. Man kann eine beliebige Anzahl solcher Bibliotheken angeben, aus denen er dann alles nimmt, was er sonst nicht kriegen kann.

Rein technisch erfolgt die Deklaration der zu untersuchenden Bibliotheken in der Kommandozeilenversion mit der Option „-l" und in der integrierten Umgebung mit einem entsprechenden Eintrag im entsprechenden Symbol des Linker-Dialogfensters. Auch in derMakedatei kann man die gewünschten Bibliotheken in der zweiten Zeile (der Linker-Anweisung mit „CPPC" am Anfang) angeben.

4.5 Initialisierungen und das große Aufräumen

In 4.1 wurde schon mit einem gewissen Stolz darauf hingewiesen, daß der MaxonC++ Linker als extraordinäres Feature auch Module initialisieren kann. Da fragt sich der geneigte Leser natürlich, woher dieses Wunderprogramm weiß, was wie initialisiert werden soll.

Die Lösung dieses Problems ist ganz einfach: Wenn der Linker ein Symbol findet, das mit „_INIT_x_" anfängt - wobei „x" eine Ziffer zwischen 0 und 9 ist - weiß er, daß diese Funktion im Startup-Code aufgerufen werden soll. Umgekehrt werden Funktionen mit dem Anfang „_EXIT_x_" am Programmende aufgerufen. Die Ziffer „x" gibt in beiden Fällen die Priorität der Initialisierung bzw. des Cleanups an: Initialisierungen mit der Nummer 0 werden als erste aufgerufen, die mit 9 als letzte. Bei den „_EXIT_"-Funktionen ist die Reihenfolge umgekehrt. Wenn der Compiler ein Modul erzeugt, das initialisiert werden muß, erzeugt er eine Initialisierungsfunktion mit der Priorität 8, und für eventuelle Destruktoraufrufe wird entsprechend eine Exit-Funktion gleichen Vorrangs generiert.

Der genaue Name der Init- und Exit-Symbole ist belanglos, sie sollten nur im Programm möglichst eindeutig sein. Deshalb kodiert der Compiler den Namen der Objektdatei in die Funktionsnamen, während in den fertigen Libraries (Sie erinnern sich: alle in Assembler geschrieben) möglichst chaotische Namen gewählt wurden, die bestimmt nirgendwo sonst vorkommen, z. B. „_INIT_0_files_o_Yeah_Initialise_me_Babe" im Modul „files.o".

Der Linker erzeugt selbstständig einen kleinen Startup-Code, der erst sämtliche Initialisierungen in der Reihenfolge ihrer Priorität aufruft und dann in den eigentlichen Startup-Code, der für gewöhnlich aus dem Bibliothksmodul „startup.o" eingebunden wird, einspringt. Außerdem generiert der Linker stets eine Funktion, in der sämtliche Exit-Funktionen aufgerufen werden. Die Standard-Funktion „exit" springt dieses Label an.

4.6 Mit und ohne Startup-Code

Manchmal will man den oben beschriebenen Startup-Code nicht haben, zum Beispiel, weil man eine Shared Library programmieren will oder weil einem der Startup-Code einfach nicht paßt. Deshalb gibt es einige Linkeroptionen, die eben diesen unterdrücken.

Zuerst wäre da die schlichte CPPC-Option „-gs" oder die Einstellung „Ohne Startup-Code" im „Linker"-Dialogfenster von „MaxonC++". Bei dieser Option werden die Module des Programms in der angegebenen Reihenfolge gelinkt, die benötigten

Bibliotheksroutinen werden angehängt und als allerletzter Hunk wird eine Sparversion des Startup-Codes erzeugt. Dieser kleine Hunk enthält dann zwei parameterlose Funktionen mit den Labels „_InitModules“ und „_CleanupModules“. Diese sollte man auf jeden Fall am Programmanfang bzw. -ende aufrufen. Von C++ aus muß man sie natürlich in C-Linkage importieren, z. B. so:

```
extern "C"
{ void InitModules();
  void ExitModules();
}
```

Diese Definitionen stehen aber auch im Include-File „<linkerfunc.h>“.

Falls man, das kanonische Beispiel für Linken ohne Startup-Code, eine Shared Library implementiert, sollte **„InitModules()“** die allererste Anweisung in der Open-Routine der Library und **„CleanupModules()“** entsprechend die letzte Anweisung in der Schließroutine sein.

Für fanatische Bytezähler, aber auch für Leute, die CPPC als BLink-Ersatz verwenden möchten, hat die CLI-Version noch zwei schlauere Optionen. Zum einen gibt es da noch „-gsi“. Hier wird der kleine Zusatz-Hunk mit den Funktionen „InitModules“ und „ExitModules“ nur dann erzeugt, wenn eines der beiden Labels auch tatsächlich irgendwo im Programm referiert wird. Dabei sind rätselhafte Programmfehler natürlich schon fast vorprogrammiert, denn man sollte die beiden Funktionen eigentlich vorsichtshalber immer aufrufen, da man ja nie wissen kann, ob irgendwelche Library-Module eine Initialisierung benötigen.

Deshalb sollten Sie „-gsi“ ganz schnell wieder vergessen und stattdessen die Option „-gsw“ verwenden. Diese funktioniert ganz genau wie „-gsi“, gibt aber eine Warnung aus, wenn eine Initialisierung bzw. ein Cleanup nötig ist, aber nicht durchgeführt wird. Im Klartext: Wenn irgendein Modul eine „_INIT_x_“-Funktion besitzt und das Label „_InitModules“ nirgendwo im Programm referiert wird, gibt der Linker so etwas wie „Symbol ‚_InitModules‘ never used“ aus - für den Cleanup analog. Die Option „-gsw“ ist recht praktisch, wenn man sich recht sicher ist, daß nirgendwo im ganzen Programm einschließlich der Standard-Bibliotheken eine dynamische Initialisierung nötig ist, die entsprechenden Funktionsaufrufe deshalb wegläßt und bei Bedarf über seinen Irrtum aufgeklärt werden möchte.

Für echte Freaks ist es natürlich unter ihrer Würde, auf fertige Routinen zurückzugreifen, und so ist ihnen auch der normale Startup-Code suspekt. Es ist nicht weiter schwierig, sich einen völlig eigenen Startup-Code zu schreiben. Linkt man ein Programm ohne Startup-Code und startet es dann, wird in die erste Routine in der ersten Übersetzungseinheit eingesprungen. Es kommt dann also beim Linken wirklich auf die Reihenfolge der Module an. Durch geschickt definierte Registerparameter kann eine solche Funktion

sogar die Shell-Argumente auswerten. Zuvor sollte sie aber die Module initialisieren und im Zweifelsfall mit „GetBaseReg()“ den Zeiger auf den Small-Data-Hunk initialisieren.

Das Beispielprogramm „nostartup.c“ demonstriert dies:

```
// nostartup.c - Demo für Linken ohne Startup-Code//
// Compileraufruf: cppc nostartup.c -gs
// bzw. "Ohne Startup" im M+Plus-Linker-Requester einstellen

#include <linkerfunc.h>
#include <stream.h>

// Die erste Funktion im ersten Hunk wird angesprungen:
void f(register __d0 int len, register __a0 char *par)
// In Registern d0 und a0 werden die Shell-Parameter übergeben
// Diese Register-Belegungen bremsen die Funktion übrigens gut aus,
// deshalb sollte man in dieser Funktion nicht mehr als nötig tun.
{
  // Wichtig, wenn Small Data-Hunks existieren, und sonst auch
  // nicht verkehrt:

  GetBaseReg(); // Initialisierungen, z. B. Dos-Lib öffnen:
  InitModules();
  par[len-1]=0; // Normalerweise kein
                // Nullbyte am Ende
  // Jetzt geht alles wie immer:
  cout << "Es geht auch ohne Startup-Code!" <<
  "\nParameter-Länge: " << len << "\nParameter: " << par
  << "\n";
  // Programmende: Aufräumen, z. B. DosLibrary schlieflen:
  CleanupModules();

  // "exit()" darf ohne Startup-Code nicht benutzt werden!!!
  // Es ist nur so ein sinnvoller Return-Code möglich:
  len = 42;
}
```

5. Die CLI-Version „CPPC“

5.1 Das Konzept

Es soll angeblich tatsächlich Leute geben, denen integrierte Entwicklungsumgebung von MaxonC++ nicht gefällt. Es gibt natürlich auch gute Gründe dafür, z. B. läßt sich „MaxonC++“ schwierig in vorhandene Umgebungen integrieren. Deshalb gibt es eine CLI-Version von MaxonC++, nämlich das Programm „CPPC“.

Eigentlich ist CPPC ebenfalls ein integriertes System, denn es enthält Compiler, Assembler und Linker. Der Unterschied zum „MaxonC++“ liegt also ausschließlich in der Benutzerschnittstelle: CPPC funktioniert wie ein CLI-Befehl und wird über Kommandozeilen-Argumente gesteuert.

Das System wurde ähnlich wie die C-Compiler unter Unix konzipiert, d. h. Compilieren, Assemblieren und Linken erfolgt mit einer einzigen Anweisung (oder optional auch getrennt). Dateinamen, die auf „.o“, „.obj“ oder „.l“ enden, werden als Objectfiles interpretiert und eingelinkt, alles andere betrachtet der Compiler als Namen von Quelltexten.

Ohne besondere Optionen übersetzt der Compiler alle angegebenen Quelltexte (ohne die entsprechenden „.o“-Files zu schreiben), linkt die angegebenen Objectfiles ein und erzeugt eine ausführbare Datei, deren Name sich vom jeweils ersten Dateinamen ableitet:

```
CPPC test.o haupt.c text.c unter.c nochwas.o
```

compiliert die drei Quelltexte „haupt.c“, „text.c“ sowie „unter.c“, erzeugt aber keine Objektdateien, sondern hält das Ergebnis dieser Übersetzungen resident im Speicher. Anschließend lädt er die beiden Objektdateien „test.o“ und „nochwas.o“ dazu und erzeugt daraus eine ausführbare Datei namens „test“.

Bei kleinen Programmen, die aus nur einem Modul bestehen, ist der Compiler also ausgesprochen einfach zu bedienen: Die Anweisung

```
CPPC name.c
```

übersetzt das Programm, linkt es und erzeugt eine ausführbare Datei namens „name“. Einfacher geht es wohl nicht.

Tritt bei der Übersetzung ein Fehler auf, erfolgt eine Ausgabe der Art

```
File "Dateiname.c", Line 17 :
Error: Identifier "ths" not defined.
Type "c" to continue, "a" to abort, "?" to view:
```

Mit „c" können Sie den Fehler übergehen, „?" gibt die fragliche Stelle des Quelltexts aus. Wenn Sie „a" eingeben oder einfach Return drücken, wird der Übersetzungsvorgang abgebrochen. Falls jedoch die Fehlermeldung mit „Warning:" beginnt, ist „continue" der Default, d. h. der Compiler macht weiter, wenn Sie nur Return drücken.

CPPC ist residentfähig, auch wenn einige Shells einen Prüfsummenfehler melden. Dies ist in jedem Fall harmlos, auch wenn CPPC auf eine etwas ungewöhnliche Methode reentrant gemacht wurde: Anhand eines Semaphors sorgt eine geladene Kopie des Programms dafür, daß sie nicht zweimal gleichzeitig läuft.

5.2 Optionen

5.2.1 Allgemeines

Die oben beschriebenen Features allein sind natürlich noch etwas mager. Alle darüber hinaus gehenden Funktionen werden durch Compileroptionen gesteuert, die in der Kommandozeile angegeben werden und jeweils mit einem „-" beginnen.

Um ein wenig Ordnung in dieses umfangreiche Sammelsurium zu bringen, lassen die Optionen sich anhand ihres jeweils ersten Buchstaben in verschiedene Bereiche einteilen:

-a	Assembler-Sourcegenerierung
-b	Debugging
-c	Objektdatei
-d	Symboldefinitionen
-e	Fehlermeldungen
-g	Codegenerierung
-i	Pfade für Includes
-j	Pfade für Libraries
-l	Linker-Libraries
-m	Make
-o	Exe-Datei
-p	Parser
-w	Warnings und Workspacegröße

Bis auf wenige Ausnahmen sind Optionen stets global: Bei einer Zeile der Form

```
CPPC -x text1.c -y text2.c -z
```

werden „text1.c“ und „text2.c“ gleichermaßen mit den Optionen „x“, „y“ und „z“ compiliert. Mit anderen Worten ist die Reihenfolge von Dateinamen und Optionen egal.

5.2.2 Ergebnisse und Ausgaben

5.2.2.1 Assembler-Quelltext

Nicht immer will man aus einem Quelltext direkt eine ausführbare Datei erzeugen. Oft ist es nötig, den Übersetzungsvorgang in einer früheren Phase abzubrechen oder ein zusätzliches Zwischenergebnis zu erzeugen, z. B. ein Assemblerlisting. Dazu dient die Option „-a“.

```
CPPC test.c -a
```

erzeugt beispielsweise nicht nur die ausführbare Datei „test“, sondern auch das Assembler-Listing „test.s“. Die Option „-a“ bricht den Übersetzungsvorgang also nicht vor der Assemblierung ab, sondern erzeugt zusätzlich zu den übrigen Aktionen den Quelltext.

MaxonC++ kann bekanntlich mehrere Texte in einem einzelnen Schritt übersetzen. Die Option „-a“ ist dabei global, d. h. unabhängig davon, vor oder hinter welchem Dateinamen sie steht, wird für alle übersetzten Module eine Assemblerdatei erzeugt.

5.2.2.2 Objektdateien: -c

Die Option „-c“ unterdrückt das Linken (und damit auch die Erzeugung einer ausführbaren Datei) und erzeugt statt dessen eine Objektdatei. Auch diese Option ist, genau wie „-a“ und so ziemlich alle anderen, global:

```
CPPC file1.C -c file2.C
```

erzeugt die beiden Objektdateien „file1.o“ und „file2.o“, und daran ändert sich auch nichts, wenn man das „-c“ an eine andere Position verschiebt.

Die Option „-c“ ist der Schlüssel zur modularen Programmierung: Wenn ein Programm aus einzelnen Übersetzungseinheiten wie „f1.c“, „f2.c“ usw. besteht, übersetzt man mit

```
CPPC -c fN.c
```

immer nur die Module, die verändert wurden, und linkt das Ganze anschließend mit

```
CPPC f1.o f2.o f3.o ...
```

zu einer ausführbaren Datei zusammen.

Eher selten wird man dagegen die Option „-cx" benutzen: Dabei werden zwar auch „.o"-Files erzeugt, aber das Projekt wird trotzdem gelinkt.

5.2.2.3 Ausführbare Programmdatei: -o

Wenn der Linker erfolgreich war und eine ausführbare Datei erzeugen konnte, benennt er diese normalerweise nach dem ersten Dateinamen, der in der Anweisungszeile auftrat, indem er einfach dessen Endung wegläßt. Falls das Ihnen nicht gefällt, können Sie mit der Option „-o" einen anderen Dateinamen angeben, z. B.

```
CPPC quelle.cpp -o Ergebnis
```

5.2.3 Preprozessor

5.2.3.1 Definieren von Symbolen: -d

Mit der Option „-d" kann man „von außen" ein Preprozessor-Symbol definieren. Dabei entspricht ein Argument

```
-d NAME
```

einer Zeile

```
#define NAME
```

am Anfang des Quelltexts. Eine typische Anwendung ist eine Anweisung wie

```
CPPC text.c -d NDEBUG
```

wenn die Includedatei „assert.h" benutzt wird.

Man kann dem Symbol aber auch einen Wert zuweisen, indem man unmittelbar hinter den Bezeichner ein „=" setzt, z. B.

```
-d ANZAHL=4711
```

Als Wert ist hier aber nur ein Token erlaubt, d. h. bei

```
-d N=17+4
```

würde das „+4“ schlicht ignoriert.

5.2.3.2 Includedateien: -i

Der Pfad, über den CPPC die Includedateien sucht, ist mit „C++:include“ vor-eingestellt. Durch einen oder auch mehrere „-i“-Optionen kann dieser Pfad geändert werden: Mit der Option

```
-i CPP:inc
```

wird ausschließlich im Verzeichnis „CPP:inc“ gesucht, und bei

```
-i CPP:include -i CPP:include/amiga
```

durchsucht der Preprozessor nacheinander die beiden Verzeichnisse.

Tritt in einem Dateinamen ein „|“ auf, z. B.

```
-i ram:include|df0:include
```

werden die gelesenen Dateien automatisch vom zweiten in den ersten Pfad umkopiert, wie es am Ende von Abschnitt 2.5.2.1 beschrieben wurde.

5.2.4 Compiler-Modus: -pc und -pp

Bekanntlich läßt sich der MaxonC++ Compiler in den ANSI C-Modus umschalten. Beim CPPC dient dazu die Option „-pc“, während „-pp“ wieder in den als Default eingeschalteten C++-Modus zurückschaltet.

Im Gegensatz zu allen übrigen Optionen ist es hier erforderlich, daß der Compilermodus vor dem jeweiligen Quelltextnamen steht, z. B. übersetzt

```
CPPC -pc a.c -pp b.c
```

die Datei „a.c“ im ANSI-Modus und „b.c“ als richtiger C++-Compiler.

5.2.5 Codeerzeugung

5.2.5.1 Breakpoints setzen: -gb

Wenn die Option „-gb“ benutzt wird, setzt der Compiler in alle Schleifen und sonstige schleifenverdächtige Konstrukte eine Abfrage des Break-Bits, das z. B. bei Betätigung der Tastenkombination Ctrl-C gesetzt wird. Dadurch ist es möglich, so erzeugte Programme jederzeit bequem abzubrechen, allerdings kostet diese Option auch einiges an Laufzeit.

5.2.5.2 Datenmodell: -gd und -gD

Die Option „-gd“ schaltet auf „Small Data“, „-gD“ auf „Large Data“ (siehe 2.5.3.1). „Large“ ist Default.

Wenn das aber schon defaultmäßig so ist - wofür existiert dann die Option „-gD“? Nun, es kann ja sein, daß in den „CPPOPT“-Voreinstellungen bereits „-gd“ voreingestellt wurde. Dann brauchen Sie „-gD“, um trotzdem auf „Large Data“ zu schalten, ohne die Environment-Variable zu ändern.

5.2.5.3 Zusammenfassen von Hunks: -gh

Mit der Option „-gh“ legen Sie die Höchstgrenze in kByte fest, bis zu der der Linker gleichartige Hunks verbindet (siehe 2.5.3.4), zum Beispiel: „-gh20“ für 20 kByte. Default ist 50.

5.2.5.4 Ganzzahlige Arithmetik: -gm

Wie bereits unter 2.5.2.4.3 beschrieben, hat der 68000-Prozessor gewisse Schwierigkeiten mit der Multiplikation und Division von 32-Bit-Zahlen. Deshalb benutzt MaxonC++ dafür normalerweise korrekt rechnende Bibliotheksfunktionen, die aber recht langsam sind.

Wenn Sie das Flag „-gm“ setzen, benutzt CPPC trotzdem die normalem Rechenoperationen des Prozessors.

5.2.5.5 Optimierungen: -go und -gO

Die zusätzlichen Optimierungen (2.5.2.4.8) werden mit der Option „-go" aus- und mit „-gO" eingeschaltet. Defaultmäßig sind sie aus.

5.2.5.6 Linken ohne Startup-Code: -gs, -gsi und -gsw

Wie in Abschnitt 4.6 beschrieben, läßt „-gs" beim Linken den Startup-Code weg, generiert dafür aber zwei explizit aufrufbare Funktionen „_InitModules" und „_CleanupModules". Bei „-gsi" werden diese nur erzeugt, wenn sie irgendwo im Projekt referiert werden, und „-gsw" gibt zusätzlich eine Warnung aus, wenn eine der beiden Funktionen nicht leer ist, aber trotzdem nirgendwo aufgerufen wird.

5.2.6 Debugger

5.2.6.1 Maxon Debugger: -b und -bi

Wenn Sie den Maxon Debugger besitzen, können Sie mit der Option „-b" die notwendigen Debugger-Dateien („.mdbi" und „.mdbg") erzeugen lassen. Die Option „-bi" schreibt zusätzlich ein Icon für die „.mdbg"-Datei.

5.2.6.2 Symbolischer Debugger: -bs

Die Option „-bs" erzeugt Symboltabellen-Hunks für symbolische Debugger (siehe 2.5.2.4.5).

5.2.7 Warnungen und Fehler

5.2.7.1 Alternative Fehlerbehandlung: -e und -en

Um die Fehlerbehandlung zu mechanisieren, z. B. auch in einem fremden Editor die Fehlerstellen automatisch anspringen zu können, bietet MaxonC++ die Option „-e". Dabei werden Fehlermeldungen in der Form

```
<Dateiname>, Line <Zeilen> / <Spalte>:
<Fehlermeldung>
```

ausgegeben, und die Abfrage „Type ‚c' to continue..." entfällt ersatzlos. Der Compiler bricht spätestens nach der zehnten Warnung oder Fehlermeldung ab. Diesen Wert können Sie verändern, indem Sie die Anzahl direkt an die Option „-e" anhängen, z. B.

```
CPPC totalfalsch.c -e20
```

Nach 50 Fehlermeldungen weigert MaxonC++ sich aber auf jeden Fall weiterzumachen.

Die Option „-en" ist identisch mit „-e", unterdrückt aber zusätzlich noch diverse Statusausgaben.

5.2.7.2 Optionale Warnungen: -w

Die unter 2.5.2.5 beschriebenen, zusätzlichen Warnungen gibt es natürlich auch beim CPPC. Man schaltet eine Warnung ein, indem man ihren Kennbuchstaben GROSS hinter das „-w" setzt, während ein kleiner Buchstabe eine Warnung ausschaltet.

Dies sind die Warnungen und ihre Kennbuchstaben:

a (Assignment): Operator „=" taucht in Bedingung auf - vielleicht der bekannte „=="-Anfänger-Fehler?

c (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. double -> int

e (Empty Statement): Anweisung hat keinen Code, z. B. „42;"

f (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)

n (Nested comments): „/*" in Kommentar

o (Old-Style): Über Parameterlisten im alten ‚K&R'-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)

p (Pragma): Unbekanntes „#pragma"

r (Return): Funktion mit Rückgabewert hat kein „return"

t (Temporary): Temporäres Objekt wurde eingeführt, um ein nicht-"const"-Referenz zu initialisieren - was seit dem 2.0-Standard eigentlich sogar ein Error ist.

v (Variable): Lokale Variable wird nie benutzt bzw. benutzt, aber nie initialisiert.

Defaultmäßig sind „-wEPRT“ aktiviert. Um z. B. Warnungen bei fehlenden Prototypen ein- und solche bei unbekannten „#pragma“ auszuschalten, setzen Sie einfach die Option „-wFp“ oder alternativ die beiden Optionen „-wF -wp“.

5.2.8 Linker

5.2.8.1 Library-Pfade: -j

Der Pfad, über den der Linker die Bibliotheken lädt, ist mit „C++:lib“ voreingestellt und kann mit der Option „-j“ verändert werden, z. B. „-j dh1:cpp/lib“. Genau wie bei der Option „-i“ (5.2.3.2) kann auch hier ein mit „|“ getrennter Doppelpfad wie z. B „ram:clib|C++:lib“ angegeben werden, um die benutzten Dateien automatisch auf ein schnelleres Speichermedium umkopieren zu können. Im Gegensatz zu den Include-Pfaden kann für Libraries aber immer nur ein Pfad bzw. Doppelpfad definiert werden.

5.2.8.2 Linker-Libraries: -l

Um Linker-Libraries (2.5.3.3) einbinden zu lassen, setzt man ihren Namen einfach hinter die Option „-l“, z. B. wird bei

```
CPPC ami.c -lamiga
```

die „amiga.lib“ eingebunden. Hinter dem „-l“ darf optional auch ein Leerzeichen stehen, und natürlich kann man auch mehrere Libraries einbinden.

5.2.9 Die integrierte Projektverwaltung: -m

Das Make-Utility gibt es nicht nur im der integrierten Umgebung „MaxonC++“, sondern auch im CPPC. Man übersetzt anhand eines Makefiles, indem mn die Option „-m“ benutzt, z. B.

```
CPPC -m makefile
```

Die Make-Datei muß die in Abschnitt 3.3 beschriebene Form besitzen. Man kann zusätzliche globale Optionen für die Übersetzung angeben, z. B.

```
CPPC -m makefile -w80 -gO -gd
```

etc. Namen zusätzlicher Text- und/oder Objektdateien werden im Make-Modus in der Kommandozeile naheliegenderweise nicht akzeptiert.

5.2.10 Arbeitsspeicher: -w

MaxonC++ alloziert zwar den Speicherplatz für die meisten internen Angelegeheiten automatisch und dynamisch, aber bei der Codegenerierung wird trotzdem ein zusammenhängender Speicherbereich benötigt, dessen Größe mit 50 kByte voreingestellt ist. Sollte bei einem umfangreichen Programm einmal eine Meldung wie

```
Workspace overflow!
```

ausgegeben werden, sollten Sie die Größe dieses Speicherbereichs mit der Option „-w" behutsam erhöhen, z. B. „-w80" für 80 kByte.

5.3 Voreinstellungen mit „CPPOPT"

Wenn es Ihnen zu mühsam ist, bei jedem Compileraufruf Ihre Lieblings-Optionen von Hand einzutippen, können Sie sich natürlich in einer geeigneten Shell ein passendes Alias definieren, z. B. haben Sie sich durch

```
alias cc2 "CPPC -pc -w100 "
```

einen C-Compiler mit großem Workspace geschaffen. Eleganter geht dies aber mit der Environment-Variablen „CPPOPT", die aber erst ab Betriebssystemversion 2.0 unterstützt wird:

```
setenv CPPOPT -pc -w100
```

stellt die beiden genanntem Optionen als Default ein. Die Variable wird so behandelt, als würde ihr Wert an den Anfang der Kommandozeilenargumente gestellt. Deshalb können Sie hier theoretisch nicht nur irgendwelche Optionen wählen, sondern z. B. auch mit

```
setenv CPPOPT test.c
```

zusätzliche Quelltexte oder Objektdateien angeben. Dies dürfte aber nicht unbedingt sinnvoll sein.

5.4 Zusammenfassung

Es folgt noch einmal eine kurze, alphabetisch geordnete Übersicht über alle CPPC-Optionen:

-a erzeugt für jedes übersetzte Textfile einen Assemblerquelltext.

-as C-Quelltext wird in Assembler-Source als Kommentar eingefügt.

-b Debug-Info-Datei erzeugen.

-bi Wie „-b", aber Icon für „.mdbg"-File erzeugen.

-bs Hunk „Symbol" für symbolischen Debugger erzeugen.

-c Die Quelltexte werden compiliert und die entsprechenden „.o"-Dateien erzeugt. Das Programm wird nicht gelinkt.

-cx Wie „-c", aber das ganze Projekt wird trotzdem gelinkt und ein Executable (daher das „x") erzeugt.

-d NAME
definiert ein Preprozessor-Symbol. Optional kann unmittelbar hinter den Namen ein „=" und der gewünschte Wert (nur ein Token) gesetzt werden, andernfalls ist das Symbol als „leer" definiert.

-e[N] Fehlermeldungen werden ausgegeben, ohne daß Interaktion des Benutzers verlangt wird. Spätestens nach dem „N"-ten Fehler wird abgebrochen (Default ist 10).

-gb Breakpoints setzen (defaultmäßig OFF).

-gd Datenmodell: Small Data.

-gD schaltet auf „Large Data", was auch Default ist.

-gh xxx
legt die maximale Größe (in kByte) fest, bis zu der gleichartige Hunks vom Linker zusammengefaßt werden. Default ist 50.

-gm MULS/DIVS-Operationen des Prozessors benutzen (Default: Bibliotheksfunktionen für korrekte 32-Bit-Arithmetik).

-go schaltet Optimierungen aus.

-gO schaltet Optimierungen ein.

Optimierungen bisher (neben Details):
- Einführen von Register-Variablen
- Ein optimierender Assembler-Pass mehr
- „link/unlk" werden nach Möglichkeit ausgelassen

-gs Ohne Startup-Code linken

-gsi Wie „-gs", aber Hunk mit „InitModules" und „CleanupModules" nur erzeugen, wenn diese Symbole auch irgendwo benutzt werden (praktisch, wenn man CPPC als schlichten Linker z. B. für Assembler-Projekte mißbraucht).

-gsw Wie „-gsi", aber zusätzlich Warnung ausgeben, wenn „InitModules" oder „CleanupModules" zwar nicht leer sind, aber trotzdem nirgendwo benutzt werden. Dann hat offenbar der Programmierer einer Shared Library o. ä. diese Aufrufe vergessen.

-i xxx

Der Pfad, über den CPPC die Includefiles sucht, ist mit „C++:include" voreingestellt. Durch einen oder auch mehrere „-i"-Argumente kann dieser Pfad geändert werden. Tritt in einem Dateinamen ein „|" auf, werden die gelesenen Files automatisch vom zweiten in den ersten Pfad umkopiert, z. B. „-i ram:includeldf0:include"

-j xxx

Analog zu „-i" stellt diese Option den Pfad bzw. den Doppelpfad, über den die Libraries gesucht werden, ein. Es kann hier aber nur ein Pfad bzw. ein Doppelpfad (mit „|") angegeben werden. Default ist „C++:lib".

-l xxx

Die Library „xxx.lib" wird aus dem aktuellen Libs-Verzeichnis eingebunden.

-m xxx übersetzt nach Makefile „xxx".

-o xxx legt einen anderen Namen für das Exe-File fest.

-pc schaltet auf Default-Mode „ANSI C".

-pp C++ als Default-Mode. „-pc" und „-pp" müssen stets VOR dem jeweiligen Quelltext stehen. „CPPC -pc a.c -pp b.c" übersetzt „a.c" als ANSI und „b.c" als C++.

-w Warnings ausgeben. Es kann eine beliebige Folge von Groß- und Kleinbuchstaben folgen, wobei ein GROSSER Buchstabe die Warnung einschaltet.

Folgende Warnungen gibt es:

a (Assignment): Operator „=" taucht in Bedingung auf - vielleicht der bekannte „=="-Anfänger-Fehler?

c (Conversion): Unsichere Typkonvertierung ohne Cast, z. B. double -> int

e (Empty Statement): Anweisung hat keinen Code, z. B. „42;"

f (Function prototypes): Funktion ohne Prototyp aufgerufen (nur in C sinnvoll, da in C++ sowieso ein Error)

n (Nested comments): „/*" in Kommentar

o (Old-Style): Über Parameterlisten im alten ‚K&R'-Stil meckern (nur in C sinnvoll, da in C++ sowieso ein Error)

p (Pragma): Unbekanntes „#pragma"

r (Return): Funktion mit Rückgabewert hat kein „return".

t (Temporary): Temporäres Objek wurde eingeführt, um ein nicht-"const"-Referenz zu initialisieren - was seit dem 2.0-Standard eigentlich sogar ein Error ist.

v (Variable): lLokale Variable wird nie benutzt bzw. benutzt, aber nie initialisiert.

Defaultmäßig sind „-wEPRT" aktiviert.

-wN Setzt den compilerinternen Workspace auf „N" KByte. Default sind 50 KByte, und das sollte reichen.

MAXONEDWARD

Nicht einfach nur ein Editor!

MAXON
computer

Inhaltsverzeichnis

Prolog 7

Einleitung - Setzen Sie Ihre Gefahr-O-Sensomatic-Sonnenbrille auf 8

I. Installation von Edward auf der Festplatte 9

II. Aufruf von Edward 9
1) Start von der Workbench 9
2) Start aus dem CLI 10

III. Eingabe und Bearbeitung von Texten 11
1) Die Status-Anzeige 11
2) BACKSPACE und DELETE 12
3) UndoLine-Funktion 12
4) RETURN, ENTER und TAB 12
5) Die Cursor-Tasten und der Ziffernblock 13
6) Der Schieberegler 14
7) Die Maus 14

IV. Text 15
1) Erzeugen von Texten 15
2) Wahl eines bestimmten Textes 16
3) Freigeben eines Textes 16
4) Löschen des aktuellen Textes 17
5) Drucken eines Textes 17

V. Dateioperationen 18
1) Speichern von Texten 18
2) Laden von Texten 19

VI. Blockoperationen 20
1) Cut, Copy, Paste 20
2) Spaltenblöcke 21
3) Laden, Speichern und Drucken 23
4) Einrücken 23
5) Umwandeln in Groß- bzw. Kleinbuchstaben 24

VII. Suchen und Ersetzen 25
1) Das „Find"-Formular 25
2) Das „Find & Replace"-Formular 26
3) Struktur-Check (Klammer-Check) 27
4) Compiler Error-Datei interpretieren, Fehler anspringen 28
5) Auf bestimmte Zeilen positionieren 29

VIII. Strukturieren von Texten, Lesezeichen 30
1) Erzeuge Falte, Zeige Falte, Entferne Falte, Zeige alle Falten, Schließe alle Falten 30
2) Auszeichnen von Zeilen 32
3) Cursor-Bewegung 33
4) Lesezeichen (Bookmarks) 33

IX. Das Edit-Prefs Formular 34

X. „Word-Wrap" 36

XI. Makros 37
1) Makros aufzeichnen & abspielen 37
2) Makros speichern & abspielen 38

XII. Fenster 40
1) Erzeugen von Fenstern 40
2) Wahl eines bestimmten Fensters 41
3) Schließen von Fenstern 42
4) Arbeiten mit mehreren Ansichten eines Textes 42

XIII. Verlassen des Editors 44
1) Unterbrechen der Arbeit 44
2) Verlassen des Editors 45

XIV. Iconify & HotKey 46
1) Iconify 46

2) HotKey 46

XV. Der Kommandozeilen-Interpreter 47

XVI. Globale Einstellungen 48
1) Das „Globale Einstellungen"-Formular 48
2) Das „Seitenformat"-Formular 51

XX. Benutzerdefinierte Menüs, Tastaturbelegung und HotKey 53
1) Kurze Einführung in die EBNF-Darstellung 53
2) Definition eigener Menüs 55
3) Definition der Tastaturbelegung 56
4) Definition des HotKeys 59
5) Einladen einer neuen Definition 60

XXI. ARexx - Kommunikation mit Edward 61

XXII. Zurücknehmen von Text-Änderungen 62

Anhang A) „Die Kommandos des Editors" 64
I. Eingabe und Bearbeitung von Texten 65
II. Verwalten von Texten 68
III. Dateioperationen 70
IV. Blockoperationen 71
V. Suchen und Ersetzen 74
VI. Strukturieren von Texten 76
VII. Ändern von Optionen, die den aktuellen Text betreffen 78

VIII. „Word-Wrap“ 79
XI. Makros 80
X. Fenster 81
XI. Verlassen des Editors 82
XII. Formulare 83
XIII. Der Kommandozeilen-Interpreter 84
XVI. Ändern von globalen Einstellungen 85
XV. ARexx - Kommunikation mit Edward 86
2) Laden von Texten aus Edward heraus 89
XVI. UnDo/ReDo - Zurücknehmen von Text-Änderungen 90
XVII. Verschiedenes 91

Anhang B) Fehlermeldungen. 92
1) Informationen 92
2) Allgemeine Fehlermeldungen 93
3) Fehlermeldungen mit Abfragen 97
4) Meldungen mit Abfragen 97
5) Fehler bei Betriebssystemoperationen 100
6) Eingabefehler 102
6) Fehler in den Definitionsdateien 103
7) Fehler beim Analysieren von Kommandos 105
8) Fehlermeldungen beim Auswerten von Kommandos 106
Sekundärliteratur 107

Index 108

Prolog

Edward - Der königliche Editor / Ein Roman von Thomas E. Wieger

Und es begab sich einmal vor langer, langer Zeit, in einem kleinen unbeugsamen Dorf, irgendwo in Deutschland, wo ein begnadeter (hüstel) Coder hauste und - seitdem die Stromversorgung in besagtem Dorfe einigermaßen stabil geworden war - in seinem Developers-Cave, tief drunten im Keller des finsteren Hauses, in welchem er wohnte, vor sich hinprogrammierte, daß jener Coder, dessen Namen wir hier noch nicht nennen möchten und aus rechtlichen Gründen auch nicht nennen dürfen, auf einen Gedanken kam (hört, hört).
Ein Gedanke, hoppla, wie konnte der nur Platz finden in dem mit seltsamen Bytes angefüllten Hirn des Coders. Nun, darüber wollen wir, liebe Leser, uns nicht den Kopf zerbrechen, warten wir lieber ab, was der Gedanke zu sagen hatte.
AE (der auktoriale Erzähler): „Nun sehr geehrter Gedanke, dann nehmen Sie doch bitte mit wenigen Worten Stellung zu Ihrem Inhalt."
GG (der göttliche Gedanke): „Ähm, tja, also ich habe vor, das gesamte Hirn des Coders mit einer einzigen Absicht zu überschwemmen. Er wird nur mich im Kopf haben, für andere Gedanken wird gar kein Platz mehr sein. Er kann nicht mehr schlafen, vom Essen wollen wir gar nicht erst reden. Nur noch ich werde ihn in seinem ganzen Handeln lenken. Tja, ja, lieber AE, das hätten Sie wohl nicht gedacht!"
AE : „Durchaus, sehr bemerkenswert. Aber wollen wir nicht nun unser Augenmerk auf Ihren besonderen Inhalt lenken. Bitte.."
GG : „Ja. Wie wunderbar. Mein Inhalt. Oh, dieses göttliche schulterlange braune Haar. Diese wunderbaren grünen Augen, in denen man wie in unergründlichen, tiefdunklen Gebirgsseen versinken könnte. Das sanfte und zarte Lächeln, das manchmal über ihr Antlitz huscht.. Ohh,... seufz... Huch, momentmal, lieber AE, Sie drücken mir ja die Luft ab, und was soll denn dieses Messer, Sie werden sich noch verletzen. Hilfeeee..."
AE (wischt sich das Blut von den Händen) : „Werter Leser, verzeihen Sie dieses unangenehme Intermezzo, selbstverständlich dachte der geniale Keller-Coder nicht an irgendwelche Frauen, vielleicht weiß er noch nicht einmal, was Frauen sind. Dieser Gedanke wollte sich nur auf äußerst lästige Weise in meine Geschichte drängen. Reine Prahlsucht war es, die ihn solchermaßen um unsere Gunst buhlen ließ."
Nun denn. Wenden wir unser Augenmerk nun dem wahren Gedanken zu, der fürwar nicht sonderlich originell war. Der Coder hatte sich, wie tausende Coder vor ihm, und tausende, die ihm noch folgen werden, dazu entschieden, einen Editor zu programmieren. Nun ja, sicher, keine besondere Idee. Aber da der Coder von dieser Idee vollständig besessen war (er hatte ja keine weitere Idee, und der Gedanke an die unbekannte braunhaarige Frau war auf grausamste Weise getötet worden), erstand im Laufe der folgenden Tage, Wochen und Monate ein Editor, der sich durch außerordentliche Merkmale auszeichnete.

Und so sprach eines Tages eine Stimme zu dem Coder mit tiefem Donnerhall : „Coder, ich rufe dich."
Coder : „Huch, hoppla, wasn das fürn Sample?"
Die Stimme : „Idiot, ich bin der obermächtige Guru, und du wirst mir Gehör schenken!"
Coder : „Au man..."
Die Stimme : „Ich habe dein Werk betrachtet und gesehen, daß es gut ist. Also hau' dich mal aufs Ohr. Ach noch was. Bevor dein Kind auszieht, um die Welt zu erobern, gib ihm einen Namen. Und da es der Sohn eines Königs ist, so braucht es einen königlichen Namen. Nenn es : EDWARD."
Die göttlichen Fanfaren erschallen.

Einleitung - Setzen Sie Ihre Gefahr-O-Sensomatic-Sonnenbrille auf

Hallo, ich bin der Autor dieser Anleitung und ich werde Sie mit Edward vertraut machen. Sollten Sie nur wenig Ahnung haben, so lesen Sie bitte die folgenden Kapitel sorgfältig durch. Mit kleinen Übungen und vielen Erläuterungen werden Sie alle Funktionen Edwards kennenlernen.
Am wichtigsten „don't panic", also keine Aufregung, gemeinsam werden wir Edward schon in den Griff bekommen und dafür sorgen, daß wir der „Chef" sind und nicht der Computer. Im Notfall legen Sie sich einfach ein Handtuch auf den Kopf.
Falls Sie schon etwas mehr Ahnung vor sich herschieben, dann blättern Sie halt einfach 'mal das Handbuch so durch oder noch besser, Sie starten Edward an und probieren alles aus. Einige Besonderheiten dürften auch Ihnen auffallen.

I. Installation von Edward auf der Festplatte

Um das System auf Ihrer Festplatte zu installieren, sollten Sie sich zunächst ein geeignetes Verzeichnis anlegen, beispielsweise mit dem Namen „WORK:EDWARD". Alles weitere erfahren Sie aus der „Readme"-Datei auf der Diskette. Da sich auch nach der Drucklegung dieses Handbuches noch Änderungen ergeben können, wählten wir diesen Weg.

Hinweis: Edwardwird beim Installieren des MaxonC++-Compiler-Systems automatisch mitinstalliert.

II. Aufruf von Edward

1) Start von der Workbench

2) Start aus dem CLI

Je nachdem, was Sie bevorzugen, läßt sich Edward sowohl aus dem CLI als auch von der Workbench aus starten. Für die fortgeschritteneren Anwendern ist hierbei vor allem interessant, welche Parameter schon beim Start an Edward übergeben werden können.

Hinweis: In Zusammenarbeit mit dem C++-Compiler muß der Editor nicht extra gestartet werden. Dies übernimmt der Compiler für Sie. Lesen Sie die beiden folgenden Abschnitte, wenn SieEDWARD auch unabhängig von MaxonC++ anwenden möchten.

1) Start von der Workbench

Am einfachsten starten Sie Edward durch Doppelklick auf sein Icon. Dieses befindet sich nach einer normalen Installation im Verzeichnis „Edward:" auf der von Ihnen gewählten Partition Ihrer Festplatte oder auf der Diskette.

Sie können Edward ebenfalls starten, wenn Sie auf das Icon einer mit Edward bearbeiteten Datei geklickt haben.

☞ Wie Sie Icons zu einer Datei erzeugen lassen können und was es dabei zu beachten gilt, lesen Sie im Kapitel „Globale Einstellungen", insbesondere sollten Sie hier die Bemerkungen zum „Icon Tool" beachten.

Wenn Sie mit gedrückter Shift-Taste die Icons verschiedener Dateien anwählen und daraufhin Edward durch Doppelklick starten (Shift-Taste gedrückt lassen), so lädt Edward automatisch die von Ihnen angewählten Dateien ein. Angezeigt wird Ihnen allerdings nur die letzte geladene Datei.

☞ Zum gleichzeitigen Bearbeiten verschiedener Dateien lesen Sie mehr im Kapitel IV „Text".

2) Start aus dem CLI

Normalerweise starten Sie Edward von CLI aus durch einfache Eingabe von „Edward". Falls Ihnen das zu lang ist, so benennen Sie Edward einfach um oder Sie benutzen den „Alias"-Befehl Ihrer Shell (Mehr dazu lesen Sie in der von Commodore ausgelieferten Dokumentation). Sie können selbstverständlich Namen von Dateien angeben, die nach dem Start von Edward automatisch geladen werden sollen.
Nach dem Start von Edward aus dem CLI erscheint die Meldung „Edward Process successfully detached". Das bedeutet, daß Edward von nun an unabhängig vom CLI als eigener Prozeß läuft. Sie können weiter mit dem CLI arbeiten, ohne daß dieser blockiert ist und können den CLI auch jederzeit beenden.

III. Eingabe und Bearbeitung von Texten

1) Die Status-Anzeige

2) BACKSPACE und DELETE

3) UNDO-Funktion

4) RETURN, ENTER und TAB

5) Die Cursor-Tasten und der Ziffernblock

6) Der Slider

7) Die Maus

Zu Beginn erläutere ich kurz die Bedeutung der Status-Anzeige, danach werde ich Ihnen die voreingestellte Tastaturbelegung erklären, so daß Sie alle üblichen Operationen bei der Eingabe eines Textes vornehmen können. Desweiteren werde ich noch zusätzliche Möglichkeiten vorstellen, wie Sie die Cursor-Position im Text verändern können.

1) Die Status-Anzeige

Wichtige Informationen über den gerade bearbeiteten Text erhalten Sie auf einen Blick, wenn Sie die Titelzeile eines momentan aktiven Text-Fensters von Edward betrachten. Ganz links erscheint dort immer der Name des gerade bearbeiteten Textes. Anschließend folgen zwei Zahlenangaben, die erste gibt die aktuelle Zeile an, die zweite die Spaltenposition innerhalb dieser Zeile.
Anschließend folgen vier durch Komma getrennte Buchstaben. Diese geben an, ob bestimmte Optionen für den aktuellen Text aktiv sind, dann ist der entsprechende Buchstabe groß, oder nicht (der Buchstabe ist dann logischerweise klein).
Das „c" ganz links gibt an, ob der Text seit dem Einladen oder nach dem letzten Abspeichern verändert wurde (großes „C" falls dies der Fall ist).
Ein großes „W" bedeutet, daß der Text im „Word-Wrap"-Modus bearbeitet wird.
Ein „O" gibt an, daß nun der „OverWrite"-Modus gilt und ein großes „L" bedeutet, daß der Text momentan ge-"locked" ist. Ge-"locked" bedeutet, der Text kann momentan nicht verändert werden. Dies ist zum Beispiel der Fall, wenn Sie den entsprechenden Text drucken lassen.

☞ Mehr zu den oben erwähnten Modi lesen Sie in den Kapiteln IX. „Das Edit-Prefs Formular“ und X. „Word-Wrap“.

2) BACKSPACE und DELETE

Die Tasten BACKSPACE und DELETE sind wie üblich belegt. Mit CTRL-DELETE können Sie die komplette aktuelle Zeile löschen.

3) UndoLine-Funktion

Edward bietet die Möglichkeit das Löschen einer Zeile oder Änderungen einer Zeile rückgängig zu machen, falls Sie nicht schon eine weitere Zeile verändert haben.

- Geben Sie zum Test folgendes ein : „Der dumme Detlev demoliert die Damentoilette“. Fahren Sie nun mit dem Cursor auf „Detlev“ und ändern Sie den Namen in „Dödel“. Drücken Sie nun CTRL-U. Daraufhin erscheint statt „Dödel“ wieder „Detlev“.
- Löschen Sie nun die komplette Zeile mit CTRL-DELETE. Mit CTRL-U erhalten Sie die komplett gelöschte Zeile wieder zurück.

4) RETURN, ENTER und TAB

Falls Sie RETURN drücken und der Cursor befindet sich innerhalb einer Zeile, so wird die Zeile an dieser Position aufgespalten. Ähnliches gilt für ENTER. Drücken Sie RETURN oder ENTER am Anfang einer Zeile, so wird vor dieser eine leere Zeile eingefügt. Drücken Sie RETURN oder ENTER am Ende einer Zeile, so wird nach dieser Zeile eine leere Zeile eingefügt. Der Unterschied zwischen RETURN und ENTER ist, daß ENTER die neu erzeugte Zeile entsprechend der vorigen Zeile einrückt. Dies ist ein besonders für Programmierer nützliches Verhalten, das das strukturierte Programmieren unterstützt.
Mit der TAB-Taste fügen Sie ein Zeichen in den Text ein, daß den Platz von 3 Leerzeichen einnimmt. Dies dient ebenfalls der besseren Strukturierung von Texten und Programmen.

☞ Die Veränderung der TAB-Weite wird im Kapitel IX. über das Edit-Prefs-Formular erklärt.

Um die oben erklärten Funktionen von TAB und ENTER zu testen, können Sie folgendes ausprobieren.

- Drücken Sie dreimal die TAB-Taste. Wie Sie an der rechten Zahl in der Titel-Leiste des Fensters sehen, befindet sich der Cursor nun an der Position 10 im Text. Geben Sie nun „for(i=10; i; i--)“ oder was Ihnen beliebt ein.
- Drücken Sie nun die ENTER-Taste. Sie befinden sich nun in einer neuen Zeile und der Cursor befindet sich wieder auf Position 10, also auf derselben Einrückungsstufe wie die vorige Zeile.

5) Die Cursor-Tasten und der Ziffernblock

Wenn Sie die CURSOR-Tasten ganz normal drücken, also nicht in Kombination mit ALT, CTRL oder SHIFT, so bewegen Sie den Cursor wie gewohnt. Dies ist ebenfalls mit den Tasten des Ziffernblock möglich. Diese müssen Sie allerdings in Kombination mit der ALT-Taste drücken (ALT- 8, 4, 6, 2).

In Verbindung mit ALT bewegen Sie sich mit CURSOR-Left und CURSOR-Right um ein Wort vorwärts oder rückwärts. Mit CURSOR-Up und CURSOR-Down geht es jeweils vier Zeilen nach oben bzw. unten.

Mit SHIFT bewegen Sie den Cursor mit CURSOR-Left und CURSOR-Right an den Anfang bzw. das Ende der Zeile. In Verbindung mit CURSOR-Up und CURSOR-Down bewegen Sie sich jeweils um fast eine Bildschirmseite nach oben bzw. unten.
Mit ALT-PgUp (ALT-9) und ALT-PgDn(ALT-3) des Ziffernblocks bewegen Sie den Cursor auf dieselbe Weise.

Mit ALT-Home (ALT-7) und ALT-End (ALT-1) des Ziffernblock springen Sie an den Anfang bzw. das Ende des Textes.

Benutzen Sie die Tasten „8“ und „2“ des Ziffernblocks in Kombination mit der linken AMIGA-Taste, so können Sie den Text um jeweils eine Zeile nach oben oder unten scrollen. Dabei bleibt der Cursor an seiner Position, und der Text verschiebt sich unter ihm in beide Richtungen.

6) Der Schieberegler

Sie können die Position des Cursors im Text ebenfalls mit dem Slider an der rechten Seite des Textfenster bestimmen. Wenn Sie vor oder hinter den Slider klicken, bewegen Sie den Cursor immer um eine Seite vor oder zurück.

7) Die Maus

Es ist zusätzlich möglich, den Cursor mit der Maus an eine bestimmte Position zu bewegen. Dazu bewegen Sie einfach den Mauspfeil auf die gewünschte Position und drücken die linke Maustaste. Solange Sie die linke Maustaste gedrückt halten, folgt der Cursor dem Mauspfeil. Bewegen Sie sich an den oberen oder unteren Rand des Fensters, so scrollt der Text in die entsprechende Richtung. Die Geschwindigkeit, mit welcher dieser Vorgang abläuft, hängt davon ab, ob sich der Mauspfeil ganz rechts - dies entspricht einer sehr langsamen Bewegung - oder ganz links - dies entspricht einer sehr schnellen Bewegung, - befindet.

IV. Text

1) Erzeugen von Texten

2) Wahl eines bestimmen Textes

3) Freigeben eines Textes

4) Löschen des aktuellen Textes

5) Drucken des aktuellen Textes

Ein Text besteht einfach aus einer oder mehreren Zeilen, die Sie eingeben oder als Datei von Diskette oder Festplatte geladen haben.

☞ Das Laden von Dateien (=Texten) finden Sie im Kapitel V. „Dateioperationen." Mit Edward können Sie fast beliebig viele Texte im Speicher halten, zwischen diesen umschalten und (natürlich :-)) die Texte bearbeiten.

1) Erzeugen von Texten

- Starten Sie Edward.

Edward hat automatisch alles Nötige unternommen, damit Sie sofort einen Text oder ein Programm eingeben können.

- Tippen Sie nun „Mein 1. Text mit Edward" ein.

- Wir werden jetzt Edward mitteilen, daß wir einen weiteren Text bearbeiten wollen. Dazu wählen Sie aus dem „Text"-Menü den Eintrag „Neu". Das Fenster von Edward ist daraufhin wieder leer. Das ist richtig, denn Sie befinden sich jetzt im zweiten Text, für den Sie eben Speicher reserviert haben.

- Tippen Sie nun „Mein 2. Text mit Edward" ein.

Mit dem oben beschriebenen Menüpunkt können Sie jederzeit Edward mitteilen, daß Sie einen weiteren Text bearbeiten wollen. Verfahren Sie wie oben beschrieben und erzeugen Sie einen dritten Text und geben daraufhin „Mein 3. Text mit Edward" ein.

2) Wahl eines bestimmten Textes

Um wieder zurück zum ersten Text zu gelangen, gibt es verschiedene Möglichkeiten.

- Wählen Sie den Eintrag „Vorheriger" aus dem Text-Menü. Edward zeigt Ihnen nun die Zeile „Mein 2. Text mit Edward" an. Mit „Vorheriger" wählen Sie immer den vorigen Text aus.

- Wählen Sie erneut den Eintrag „Vorheriger" aus dem Text-Menü. Edward zeigt Ihnen nun die Zeile „Mein 1. Text mit Edward" an. Wir befinden uns also jetzt, wie gewünscht, wieder im ersten Text.

- Wählen Sie noch einmal den Eintrag „Vorheriger" aus dem Text-Menü. Edward zeigt Ihnen die Zeile „Mein 3. Text mit Edward" an. Vor dem ersten Text kann natürlich kein weiterer liegen. Darum wählt Edward in diesem Fall den letzten Text. Sie können also mit „Vorheriger" die Texte der Reihe nach anwählen.

Genauso können Sie durch Wählen von „Nächster" im „Text"-Menü den folgenden Text anwählen. Haben Sie gerade den letzten Text bearbeitet, so wählt Edward in diesem Fall den ersten Text.

- Eine direkte Wahl des Textes ist ebenfalls möglich. Möchten Sie den zweiten Text bearbeiten, dann wählen Sie einfach „Text 2" aus dem „Text"-Menü.

3) Freigeben eines Textes

Nach der Bearbeitung eines Textes (und dem Abspeichern desselben) können Sie Edward mitteilen, daß Sie diesen Text freigeben möchten.

- Wählen Sie „Löschen" aus dem „Text"-Menü. Edward zeigt Ihnen daraufhin die Zeile „Mein 3. Text mit Edward" an. Der zweite Text wurde von Edward freigegeben. Nach der Freigabe eines Textes versucht Edward, den folgenden zu wählen. Haben Sie den letzten Text freigegeben, dann wählt Edward den vorigen Text für die weitere Bearbeitung aus.

☛ Hinweis:
Sollten Sie aus Versehen die Freigabe eines Textes eingeleitet haben, der noch nicht abgespeichert wurde, so erscheint ein Dialogfenster mit der Frage, ob Sie diesen Text „vergessen" wollen. Geben Sie „Nein" an, so wird die Operation abgebrochen, wählen Sie „Ja", dann gehen alle Änderungen, die an diesem Text vorgenommen wurden, verloren!

4) Löschen des aktuellen Textes

Mit dem Eintrag „Entfernen" aus dem „Text"-Menü können Sie alle Zeilen des aktuellen Textes löschen und danach einen neuen Text eingeben. Der Text wird dadurch aber nicht freigegeben.

☛ Hinweis:
Sollten Sie aus Versehen die Löschung eines Textes eingeleitet haben, der noch nicht abgespeichert wurde, so erscheint ein Dialogfenster mit der Frage, ob Sie diesen Text „vergessen" wollen. Geben Sie „Nein" an, so wird die Operation abgebrochen, wählen Sie „Ja", dann gehen alle Änderungen, die an diesem Text vorgenommen wurden, verloren!

5) Drucken eines Textes

Mit dem Eintrag „Drucken" aus dem „Projekt"-Menü teilen Sie Edward mit, daß er den Text, den Sie momentan bearbeiten, ausdrucken soll. Während Edward diesen Text druckt, können Sie weiterhin andere Texte bearbeiten und desweiteren alle Funktionen von Edward benutzen, die den aktuellen Text nicht verändern. Sie können also, während Edward Ihren Text druckt, weiterhin in diesem den Cursor positionieren oder z.B. Blöcke markieren und kopieren. Das Einfügen oder Ausschneiden von Blöcken ist dagegen nicht möglich.
Während des Druckvorgangs erscheint ein kleines Fenster. Wenn Sie den Druckvorgang unterbrechen wollen, so schließen Sie einfach dieses Fenster.

☞ Das Format, in welchem eine Textseite gedruckt werden soll, können Sie auf vielfältige Weise variieren. Lesen Sie dazu mehr im Kapitel XVI. „Globale Einstellungen".

☛ Wichtig:
Edward gibt den Text an den Drucker aus, den Sie mit den Preferences der Workbench gewählt haben. Sollten Sie hiermit nicht klarkommen, so schauen Sie im Handbuch zur Workbench nach, das zu jedem Amiga mitgeliefert wird.

V. Dateioperationen

1) Speichern von Texten

2) Laden von Texten

Hinter dem hochtrabenden Titel „Dateioperationen" verbergen sich Funktionen zum Abspeichern und Laden von Texten.

1) Speichern von Texten

Nachdem Sie im ersten Kapitel gelernt haben, wie man einen Text eingibt und bearbeitet, zeige ich Ihnen nun, wie man einen solchen Text auf Diskette oder Festplatte abspeichert und wie man den Text von dort zur Bearbeitung wieder in den Editor bekommt.

- Geben Sie nun einen beliebigen Text ein, z.B.: „Alles Seiende hat sein Sein im Nicht-Sein seines Gegensatzes."

- Wählen Sie aus dem „Projekt"-Menü den Eintrag „Speichern als...". Es erscheint daraufhin ein Dateiauswahlfenster, in dessen Titelzeile „Text speichern" steht. Speichern Sie den Text unter dem Namen „Test.TXT" ab.

Normalerweise wird beim Abspeichern eines Textes für diesen ein Pictogramm erzeugt, so daß er auch auf der Workbench sichtbar wird. Desweiteren werden einige Zusatzinformationen, die den Text betreffen mitabgespeichert (unter anderem die Tabulatorweite, Textfalten, etc.). Besonders interessant ist hierbei, daß Edward auch die aktuelle Cursorposition speichert. Das bedeutet, wenn Sie einen Text erneut einladen, dann befindet sich der Cursor an derselben Stelle, an der Sie zuletzt gearbeitet haben.

☞ Das Erzeugen des Pictogrammes und das Speichern der Zusatzinformationen können Sie natürlich auch abstellen. Lesen Sie hierzu das Kapitel XVI. „Globale Einstellungen".

☛ Hinweis:
Unter Umständen erscheint ein „System Request" mit dem Hinweis, das Ihre Diskette schreibgeschützt ist. Sorgen Sie dann dafür, das der Schreibschutz entfernt wird oder nehmen Sie eine andere Diskette.

☛ Wichtig:
Die Bedeutung des Abspeicherns kann gar nicht oft genug betont werden. Alle Veränderungen, die Sie seit dem letzten Abspeichern vorgenommen haben, sind verloren, wenn ein System-Absturz auftritt, der Strom ausfällt oder sonstige unvorhersehbare Ereignisse eintreten. Sie können durch eine spezielle Einstellung dafür sorgen, daß Edward in bestimmten Zeitabschnitten Ihre Texte automatisch abspeichert.

☞ Prefs/Autosave

2) Laden von Texten

Nachdem Sie nun einen Text abgespeichert haben, zeige ich Ihnen im Folgenden wie Sie diesen Text wieder einladen.

- Wählen Sie aus dem „Text"-Menü den Eintrag „Entfernen". Das Textfenster ist nun leer. Alle Textzeilen wurden freigegeben. Dies ist nur für unser Beispiel notwendig.

- Wählen Sie nun den Eintrag „Öffnen" aus dem „Projekt"-Menü. Es erscheint wie vorher beim Abspeichern der Dateiauswahlfenster. In diesem Fall steht in seiner Titelzeile aber „Text laden". Laden Sie den Text „Test.TXT" wieder ein.

☛ Hinweis:
Wenn Sie einen Text einladen, wird dieser über den Inhalt Ihres aktuellen Textes geladen. Wenn Sie dies vermeiden wollen, so sollten Sie zuerst einen neuen Text erzeugen, bevor Sie „Öffnen" wählen. Sollte der aktuelle von Ihnen bearbeitete Text noch nicht gespeichert worden sein, so erscheint ein Dialog-Fenster, das Sie auf diesen Umstand aufmerksam macht und mit dem Sie das Überladen dieses Textes verhindern können.

VI. Blockoperationen

1) Cut, Copy, Paste

2) Spaltenblöcke

3) Laden, Speichern und Drucken

4) Einrücken

5) Umwandeln in Groß- bzw. Kleinbuchstaben

Größere Teile eines Textes umgruppieren oder vervielfältigen zu können, ist eine der wichtigsten Fähigkeiten eines Editors. Edward bietet Ihnen auch auf diesem Gebiet alles Nötige und einiges darüber hinaus.
Ein besonders markierter Bereich eines Textes wird im allgemeinen als „Block" bezeichnet. Daraus leitet sich auch der Name dieses Kapitels ab.

1) Cut, Copy, Paste

Hinter den Begriffen „Ausschneiden", „Kopieren" und „Einfügen" verbergen sich die elementaren Blockoperationen.
Mit „Ausschneiden" schneiden Sie einen markierten Block aus Ihrem Text aus und speichern ihn in einem Zwischenspeicher ab. „Kopieren" kopiert den Block nur in den Zwischenspeicher, ohne ihn auszuschneiden und „Einfügen" fügt den Block ab der aktuellen Cursor-Position in den Text ein.

Um eine Blockoperation ausführen zu können, müssen Sie zuerst einmal einen Block markieren. Dies geschieht indem Sie „Markieren" aus dem „Block"-Menü wählen. Wenn Sie nun den Cursor bewegen, so werden Sie feststellen, daß der gesamte Textbereich zwischen der Position an der sich der Cursor momentan befindet und dem Ort, an dem sich der Cursor befand, bevor Sie „Markieren" gewählt haben, invertiert dargestellt wird. Dieser invertiert dargestellte Bereich ist der sogenannte markierte „Block". Alle Blockoperationen wirken sich auf diesen Bereich aus.
Wenn Sie erneut „Markieren" wählen, wird der gesamte Text wieder normal dargestellt. Der markierte Block wurde damit vergessen.

- Für die folgenden Schritte sollten Sie nun einen etwas längeren Text einladen.

- Positionieren Sie nun den Cursor in die Mitte irgendeiner Zeile am Anfang des Textes und wählen Sie danach „Markieren" aus dem „Block"-Menü.

- Fahren Sie nun mit dem Cursor in irgendeine Zeile, die weiter unten liegt. Sie sehen nun einen invertiert dargestellen Bereich.

- Wählen Sie „Ausschneident" aus dem „Block"-Menü. Der markierte Bereich verschwindet daraufhin. Der Cursor befindet sich nun auf derselben Position, an der er sich befand, bevor Sie „Markieren" gewählt hatten. Rechts vom Cursor befindet sich der Rest der Zeile, auf der sich der Cursor befand, bevor Sie „Ausschneiden" gewählt hatten.

Um nun die ursprüngliche Gestalt des Textes wiederherzustellen, können Sie den ausgeschnittenen Bereich wieder in den Text einfügen. Dazu dürfen Sie den Cursor nicht bewegen.

- Wählen Sie „Einfügen" aus dem „Block"-Menü. Der Cursor befindet sich nun am Ende des eingefügten Bereiches. Ihr Text hat wieder seine ursprüngliche Gestalt.

Sie können jederzeit an anderen Positionen im Text eine Kopie des ausgeschnittenen Bereiches einfügen, indem Sie einfach „Paste" anwählen.
Wenn Sie einen bestimmten Textbereich nur kopieren wollen, so markieren Sie den Bereich, wählen „Kopieren", positionieren den Cursor an die gewünschte Stelle und wählen danach „Einfügen".

☛ Zum Zwischenspeichern von Textausschnitten bedient sich Edward des Zwischenspeichers. Dies ist eine genormte Ablage, der sich alle Programme, die für den Amiga entwickelt werden, bedienen sollten. Falls Sie zum Beispiel ein Desktop-Publishing Programm besitzen, aber nicht auf die komfortablen Editier-Möglichkeiten Edwards verzichten wollen, so können Sie sich des Zwischenspeichers bedienen, um Texte zwischen Edward und dem DTP-Programm auszutauschen. Dabei ist natürlich Voraussetzung, daß das DTP-Programm ebenfalls den Zischenspeicher unterstützt.

2) Spaltenblöcke

Eine Besonderheit im Umgang mit Blöcken bietet Ihnen Edward mit den Spaltenblöcken. Haben Sie zum Beispiel eine Tabelle mit mehreren Spalten eingegeben und Sie

wollen nun eine der Spalten kopieren oder vertauschen, dann bewerkstelligen Sie dies unter Zuhilfenahme von Edwards Spaltenblöcken.

- Geben Sie folgende Tabelle ein

Film	Bewertung	Hauptdarsteller	!Test
Alien I	1-	Sigourney Weaver	!
Alien II	1	Sigourney Weaver	!
Terminator II	1++	Arnold Schwarzenegger	!
Homo Faber	1	Sam Shephard	!
Gefährliche Liebschaften	1+	John Malkovic,	!
		Glenn Close,	!
		Michelle Pfeiffer	!
The Fabulous Baker Boys	1	Michelle Pfeiffer,	!
		Jeff & Beau Bridges	!
Das Russlandhaus	2	Sean Connery,	!
		Michelle Pfeiffer	!
Harry & Sally	1	Meg Ryan	!
Moon on Earth	1	Wynona Ryder	!
Fessle mich!	1	-	!
Das Wunderkind Tate	2+	Jodie Foster	!
König der Fischer	2+	Robin Williams	!
Hook	6	Dustin Hoffmann,	!
		Robin William,	!
		Julia Roberts	!

Sie wollen nun die Spalte „Bewertung“ mit der Spalte „Hauptdarsteller“ vertauschen.

- Fahren Sie mit dem Cursor auf das B von „Bewertung“. Wählen Sie dann „vertikal markieren“ aus dem „Block“-Menü.

- Bewegen Sie nun den Cursor vor das R von „Robin Williams“ in der letzten Zeile, Spalte „Hauptdarsteller“. Wählen Sie anschließend „Ausschneiden“ aus dem „Block“-Menü. Die Spalte „Bewertung“ ist nun verschwunden, „Hauptdarsteller“ befindet sich an deren Stelle.

- Positionieren Sie den Cursor auf das Ausrufungszeichen in der ersten Zeile der Spalte „Test“. Wählen Sie nun „vertikal einfügen“ aus dem „Block“-Menü.

Daraufhin erscheint die Spalte „Bewertung“ wieder. Die Vertauschung hat funktioniert.

3) Laden, Speichern und Drucken

Sie können einen mit „Ausschneiden" oder „Kopieren" in die Zwischenablage kopierten Block auf einfache Weise als Datei abspeichern. Dazu wählen Sie einfach „Block speichern" aus dem „Bloxk"-Menü. Es erscheint der Dateiauswahlfenster und Sie können einen Namen für den Block angeben.
Wollen Sie einen so abgespeicherten Block wieder in Ihren Text einfügen, so wählen Sie „Block einlesen" aus dem „Block"-Menü. Mit dem Dateiauswahlfenster geben Sie eine Datei an. Daraufhin wird die angebene Datei in den Zwischenspeicher geladen. Sie können sie dann jederzeit mit „Einfügen" an einer beliebigen Stelle im Text einfügen.

Mit dem Menüpunkt „Block drucken" aus dem „Block"-Menü können Sie einen in die Zwischenablage gespeicherten Block ausdrucken, allerdings ohne die Ausgabe besonders zu formatieren, wie es beim Ausdruck eines ganzen Textes geschieht.

4) Einrücken

Zur Strukturierung von Texten, insbesondere von Programmen, dient das Einrücken von Textabschnitten, die damit als zusammengehöriger Abschnitt hervorgehoben werden sollen. Zur Unterstützung dieser Maßnahmen bietet Ihnen Edward die Möglichkeit, einen markierten Bereich von Zeilen einzurücken.

- Geben Sie folgendes Programmfragment ein :

```
for( i=1000; i>0; i-- )
{
  c=c0*c0+c1*c1-r;
  c++;
  r++;
}
```

Um den Bereich zwischen der öffnenden und der schliessenden geschweiften Klammer ("{","}") besser hervorzuheben, können Sie ihn einrücken.

- Bewegen Sie dazu den Cursor auf die Zeile nach der öffnenden geschweiften Klammer und wählen Sie „Mark" aus dem „Edit"-Menü.

- Fahren Sie nun mit dem Cursor an das Ende der Zeile vor der schliessenden geschweiften Klammer.

Mit CTRL-CursorLeft und CTRL-CursorRight können Sie nun den markierten Bereich um jeweils einen TAB-Schritt nach rechts oder links verschieben.

- Drücken Sie CTRL-CursorRight. Damit haben Sie den Block um eine TAB-Position nach rechts verschoben. Probieren Sie ruhig ein bißchen die Möglichkeiten aus, die sich mit dem oben beschriebenen Verfahren ergeben.

5) Umwandeln in Groß- bzw. Kleinbuchstaben

Mit zwei weiteren Funktionen können Sie alle Buchstaben eines markierten Blocks in Groß- bzw. Kleinbuchstaben wandeln lassen.

- Geben Sie „Alles Seiende entsteht ohne Grund, setzt sich aus Schwäche fort und stirbt durch Zufall." oder etwas was Ihnen mehr zusagt, ein.
- Markieren Sie einen Block, der vom ersten Buchstaben von „Seiende" bis zum letzten Buchstaben von „aus" reicht.
- Wählen Sie nun „Großbuchstaben" aus dem „Extras"-Menü. Augenblicklich werden alle markierten Buchstaben in Großbuchstaben umgewandelt.

Wenn Sie „Kleinbuchstaben" aus dem „Extras"-Menü wählen, so werden entsprechend alle Buchstaben des markierten Blocks in Kleinbuchstaben gewandelt.

VII. Suchen und Ersetzen

1) Das „Find"-Formular

2) Das „Find & Replace"-Formular

3) Struktur-Check (Klammer-Check)

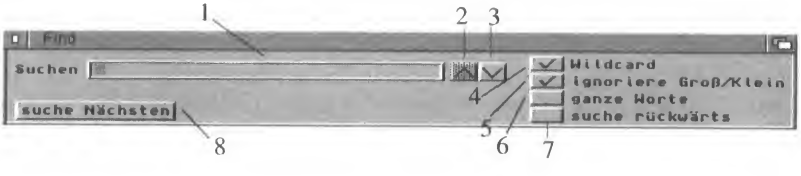
4) Compiler Error-Files interpretieren und Fehler anspringen

5) Auf bestimmte Zeilen positionieren

Suchen und Ersetzen von Zeichenfolgen in einem Text gehören zum Standardrepertoire eines Editors. Hier läßt Sie Edward natürlich nicht im Stich.
Um Ihnen das Arbeiten beim Suchen und Ersetzen zu erleichtern, bietet Edward zwei Extra-Fenster mit diversen Einstellmöglichkeiten an, über welche Sie die Arbeiten beim Suchen und Ersetzen steuern. Da Sie in diesen Fenstern gewisse Eintragungen machen können, sprechen wir im folgenden von sogenannten „Formularen".

1) Das „Find"-Formular

Zum Suchen eines bestimmten Begriffes bedienen Sie sich des „Find"-Formulars. Dieses erscheint, wenn Sie im Menü „Suchen" den Menüpunkt „Suchen ..." wählen.



Wir werden nun gemeinsam die Funktionen des „Find"-Formulars durchsprechen.

1) In diesem Bereich geben Sie die zu suchende Zeichenfolge ein. Nach dem Öffnen eines Find-Formulars ist 1) automatisch aktiv und zur Eingabe bereit. Nach dem Aufruf des „Find"-Formulars ist er sofort zur Eingabe bereit. Wenn Sie die

Eingabe mit RETURN abschliessen, so verschwindet das „Find"-Formular gleich darauf und der Cursor wird auf die erste Stelle im Text, wo diese Zeichenfolge gefunden werden konnte positioniert.
Für die Suche einer Zeichenfolge gilt es dabei Folgendes zu beachten:
Gesucht wird immer von der aktuellen Cursorposition aus. Normalerweise wird von dieser ausgehend dann in Richtung auf das Ende des Textes hin gesucht. Konnte dabei keine Zeichenfolge im Text gefunden werden, die der gewünschten entspricht, so wird die Suchrichtung umgekehrt. Jeder weitere Suchvorgang geht dann von der Cursorposition aus zum Beginn des Textes hin. Wird dort nichts mehr gefunden, so kehrt sich die Richtung erneut um.

7) Mit 7) können Sie die Suchrichtung jederzeit umkehren.

4) Ist 4) angewählt, so können beim Suchen sogenannte WildCards eingesetzt werden. Als sog. Jokerzeichen dient dabei der Stern „*". An jeder Stelle an der sich in der zu suchenden Zeichenfolge ein Stern befindet darf beim Vergleich mit einer anderen Zeichenfolge eine beliebige Teilfolge stehen.
Geben sie zum Beispiel „Co*m*e" als zu suchende Zeichenfolge ein, so wird beim Vergleich unter anderem „Commodore" oder „Commode" als gefunden erkannt.

5) Wenn Sie 5) anwählen, so wird beim Suchen nicht zwischen Groß- und Kleinbuchstaben unterschieden.

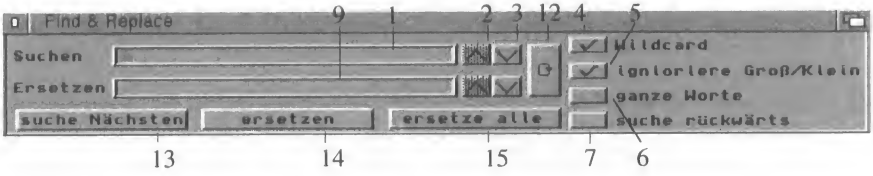
6) Wählen Sie 6), so wird nur nach vollständigen Wörtern gesucht. Wenn Sie also die Zeichenfolge „Kohl" suchen und Sie haben 6) gewählt, so gilt „Kohlsuppe" oder „Kohldampf" als nicht gefunden. Nur „Kohl „ wird gefunden und der Cursor entsprechend positioniert.

2), 3) Eine Besonderheit bietet Edward Ihnen mit 2) und 3). Sie können insgesamt 10 verschiedene Suchmuster eingeben. Mit den Pfeilen können Sie zwischen Ihnen umschalten.

8) Mit 8) positionieren Sie den Cursor automatisch auf das nächste Vorkommen der in 1) angebenen Zeichenkette.

2) Das „Find & Replace"-Formular

Mit dem „Find & Replace"-Formular können Sie auf komfortable Weise Zeichenfolgen im Text suchen und durch andere Zeichenfolgen ersetzen lassen.



Die Bedienung des „Find & Replace"-Formular ist mit der des „Find"-Formular weitgehend identisch. Dies gilt insbesondere für die durch 4) - 8) wählbaren Optionen.

1) In 1) geben Sie die zu suchende Zeichenfolge an, in 9) die Zeichenfolge durch welche 1) ersetzt werden soll. Durch das Öffnen des „Find & Replace"-Formulars wird automatisch 1) aktiviert. Nach der Eingabe in 1) wird automatisch 9) aktiviert.

2),3) 10),11) Mit 2) und 3) können Sie zehn verschieden Suchmuster anwählen, mit 10) und 11) analog zehn verschieden Zeichenfolgen durch die diese ersetzt werden sollen.

12) Als weitere Komfortsteigerung können Sie durch 12) die Such- und Ersetzungszeichenfolgen miteinander vertauschen. Falls Sie also aus Versehen zum Beispiel „Hund" suchen und durch „Katze" ersetzen ließen so können sie durch anklicken von 12) und erneutes Ersetzen das Ganze rückgängig machen.

13) Mit 13) positionieren Sie den Cursor automatisch auf das nächste Vorkommen der in 1) angebenen Zeichenkette.

9),14) Klicken Sie auf 14) dann wird von der aktuellen Cursor-Position aus die in 1) angebebene Zeichenfolge gesucht und, falls gefunden, durch die in 9) angebenene Zeichenfolge ersetzt. Danach wird automatisch das nächste Vorkommen von 1) gesucht und der Cursor darauf positioniert.

15) Mit 15) werden im gesamten Text automatisch alle Vorkommen von 1) durch 9) ersetzt. Anschließend wird der Cursor auf den Beginn des Textes positioniert.

3) Struktur-Check (Klammer-Check)

Für Programmierer (insbesondere diejenigen, die die Sprache „C" benutzen) kann der Strukur-Check sehr hilfreich sein. Mit diesem läßt sich auf einfache Weise prüfen, ob für jede öffnende Klammer in einem Ausdruck auch eine schließende Klammer vorhanden ist. Dabei werden nicht nur die gewohnten runden Klammern „(„ und „)" überprüft, sondern auf Wunsch auch „[" und „]" sowie „{" und „}".

- Geben Sie folgende Zeilen ein :

```
void main( void )
{
  int i;
  for( i=100; i>0; i-- )
  {
    if( i & 1 )
    { printf( "%d\n", i*i+((3*i)-10/i) );
    }
  }
}
```

- Positionieren Sie nun den Cursor auf die öffnende geschweifte Klammer in der zweiten Zeile.
- Nun wählen Sie „Strukturkontrolle“ aus dem „Suchen“-Menü. Der Cursor wird automatisch auf die korrespondierende schließende geschweifte Klammer in der letzten Zeile positioniert.
- Wenn Sie erneut „Strukturkontrolle“ wählen, dann wird der Cursor wieder auf die öffnende geschweifte Klammer in der zweiten Zeile gesetzt.

Die Funktion „Strukturkontrolle“ versucht immer den Cursor auf die zur Klammer unter dem Cursor korrespondierende öffnende oder schließende Klammer zu setzen. Sollte dies nicht möglich sein, so meldet Edward „Structures do not match!“. Somit können Sie bei stark verschachtelten Funktionsblöcken in Ihren Programmen auf einfache Weise überprüfen, ob Sie zu jeder öffnenden auch eine schließende Klammer gesetzt haben und Sie müssen nicht erst den Compiler bemühen, um solche leicht zu vermeidenden Syntax-Fehler zu entdecken.
Mit den runden Klammern in der sechsten Zeile des obigen Beispiels können Sie die Funktion „Match Bracket“ ausprobieren. Nehmen Sie ruhig einmal eine Klammer heraus und stellen Sie fest, wie Edward darauf reagiert.

4) Compiler Error-Datei interpretieren, Fehler anspringen

Eine Besonderheit von Edward ist, daß er die Fehlerdateien bestimmter Compiler auswerten kann. Sie müssen nur von Ihrem Compiler eine Fehlerdatei erzeugen lassen und diese mit Edward laden. Die dafür nötigen Funktionen können Sie zum Beispiel auf eine Taste oder ein Menü legen. Bei einem C-Compiler können Sie durch einfachen Aufruf von „CC >ErrorFile ...“ die Fehlermeldungen in die Datei „ErrorFile“ umleiten. Wenn Sie die Fehlerdatei geladen haben, so positionieren Sie den Cursor auf eine Zeile in der die Zeilennummer eines Fehlers steht. Danach wählen Sie „Gehe zu Fehler“. Daraufhin überprüft Edward, ob die Datei, zu der die Fehlerbeschreibung gehört, sich schon im Speicher befindet. Ist dies nicht der Fall, so wird die entsprechende Datei nachgeladen. Daraufhin wird der Cursor auf die Zeile positioniert, in welcher sich der angebene Fehler befindet.

☛ Wenn Sie in einer geladenen Quelldatei einige Veränderungen vornehmen und dabei Zeilen einfügen oder löschen, so kann „Gehe zu Fehler“ natürlich nicht mehr auf die richtige Zeile positionieren. Sie sollten daher immer die ungefähren Verschiebungen von Zeilen im Kopf behalten.

5) Auf bestimmte Zeilen positionieren

Mit „Gehe zur Zeile..“ öffnen Sie ein kleines Formular, in dem sofort ein Feld zur Eingabe einer Zahl bereit ist. Geben Sie die Nummer der gewünschten Zeile ein und drücken Sie „RETURN“. Daraufhin versucht Edward den Cursor auf die gewünschte Zeile zu positionieren.

Mit „Gehe zu m Anfang“ können Sie an den Anfang eines Textes mit „Gehe zum Ende“ an das Ende eines Textes gelangen.

VIII. Strukturieren von Texten, Lesezeichen

1) Erzeuge Falte, Zeige Falte, Entferne Falte, Zeige alle Falten, Schließe alle Falten

2) Auszeichnen von Zeilen

   Mark, UnMark

3) Cursor-Bewegung

4) Lesezeichen (Bookmarks)

Programme bestehen normalerweise aus Prozeduren, die in weitere kleine Einheiten strukturiert sind (oder zumindest sein sollten). Um ein Programm etwas übersichtlicher darstellen zu können, bietet Ihnen Edward das Konzept der Falten an. Desweiteren können Sie bestimmte Zeilen markieren. Mit den Cursor-Tasten lassen sich so bestimmte Strukturen einer Datei leicht anwählen. Ausserdem können wichtige Zeilen gemerkt und mit einem Tastendruck direkt angesprungen werden (Lesezeichen).

☛ Hinweis:
Die Strukturierung eines Textes wird in der zum Text gehörigen Pictogramm-Datei abgespeichert. Näheres hierzu erfahren Sie im Kapitel XVI. „Globale Einstellungen".

1) Erzeuge Falte, Zeige Falte, Entferne Falte, Zeige alle Falten, Schließe alle Falten

Hinter dem Konzept der Falten verbirgt sich eine sehr leistungsfähige Methode zur Strukturierung von Programm-Quelldateien.

- Geben Sie nun folgendes kleine Programm ein oder laden Sie die Quelldatei eines von Ihnen erstellten Programms ein.

```
void strnrcpy
(
  char *string0,
  char *string1,
  int n
)
{
  int Pos;

  Pos=strlen( string1 );
  if( Pos>n )
    Pos=Pos-n;
  else
    Pos=0;
  strcpy( string0, &string1[Pos] );
}

void strtolower
(
  register char *StringP
)
{
  register char c;
  while( (c=*StringP)!=EOS )
  {
    if( isalpha(c) )
      *StringP++=_tolower(c);
    else
      StringP++;
  }
}
```

- Bewegen Sie den Cursor auf die erste Zeile. Wählen Sie „Markieren“ aus dem „Block“-Menü. Positionieren Sie den Cursor nun unter die letzte Zeile der Prozedur „strnrcpy“.
- Jetzt wählen Sie „Erzeuge Falte“ aus dem „Falte“-Menü. Alle markierten Zeilen bis auf die erste verschwinden nun. Vor dieser erscheint das Zeichen „«“. Der Cursor wurde ebenfalls auf diese Zeile bewegt.

Damit haben Sie die erste Falte erzeugt. Alles, was vorher markiert war, steckt nun in der Falte. Nur die erste Zeile ist noch sichtbar. Ein besonderes Zeichen zeigt Ihnen den Beginn der Falte an.

- Wählen Sie nun „Zeige Falte". Daraufhin wird der Inhalt der Falte wieder sichtbar. Die Zeilen innerhalb der Falte beginnen mit dem Zeichen „|". Die letzte Zeile einer Falte wird durch das Zeichen „»" kenntlich gemacht.

- Wählen Sie erneut „Erzeuge Falte", dann wird die Falte wieder „eingeklappt" und die vorher angezeigten Zeilen verschwinden.

- Wenn Sie „Entferne Falte" anwählen, so verschwindet die Markierung der gefalteten ersten Zeile, und alle „innereren" Zeilen der Falte erscheinen wieder. Diesmal allerdings unmarkiert. Die Faltung wurde vollständig rückgängig gemacht.

☛ „Zeige Falte" und „Entferne Falte" arbeiten nur, wenn der Cursor auf die erste Zeile einer Falte positioniert wurde. „Erzeuge Falte" faltet einen markierten Block ein oder schließt eine mit „Zeige Falte" geöffnete Falte wieder.

- Falten Sie nun beide Prozeduren aus unserem Beispiel ein.
- Jetzt wählen Sie „Zeige alle Falten". Daraufhin erscheinen beide gefalteten Bereiche so, alls wäre für jeden die Funktion „Zeige Falte" aufgerufen worden.
- Durch Wahl von „Schließe alle Falten" wird das Ganze wieder rückgängig gemacht und nur die erste Zeile jeder Falte ist noch sichtbar.

Mit „Entferne alle Falten" entfernen Sie alle Falten aus Ihrem Text.

Selbstverständlich lassen sich die Falten ineinander schachteln. Das bedeutet, das Sie mehrere gefaltete Bereiche in einer übergreifenden Falte zusammenfassen können.

Mit „Zeige innere Falten" können Sie auf einen Schlag eine Falte mit all ihren inneren Falten öffnen. „Schließe innere Falten" macht dies rückgängig.
Mit „Entferne innere Falten" entfernen Sie eine Faltung inklusive all ihrer Unterfaltungen.

2) Auszeichnen von Zeilen

Bestimmte Zeilen eines Dokuments oder einer Quelldatei, die Sie mit Edward bearbeiten, können Sie als besonders wichtig kennzeichnen. Diese Zeilen können mit den in Abschnitt 3) erwähnten Methoden auf besonders einfache Weise angesprungen werden.

Eine Zeile wird durch die Tastenkombination „ALT-SHIFT-RETURN" ausgezeichnet. Die Zeile wird daraufhin mit einem führenden Ausrufungszeichen dargestellt. Wiederholen Sie diese Tastenkombination, so wird die Auszeichnung zurückgenommen, und das Ausrufungszeichen verschwindet.

3) Cursor-Bewegung

Mit zwei Tastenkombination können Sie gefalteten Zeilen sehr einfach anspringen. Mit „CTRL-CURSOR_UP" wählen Sie eine vor der aktuellen Zeile liegende Zeile an, die einen Faltenstart oder ein Faltenende darstellt. Ebenso können Sie mit „CTRL-CURSOR_DOWN" eine nach der aktuellen Zeile liegende Zeile anspringen.

Mit „ALT-SHIFT-CURSOR_UP" bzw. „ALT-SHIFT-CURSOR_DOWN" springen Sie eine vor bzw. nach der aktuellen Zeile liegende Zeile an, die Sie ausgezeichnet hatten.

4) Lesezeichen (Bookmarks)

Beim Arbeiten mit einem Text kommt es oft vor, daß man schnell verschiedene, weit auseinanderliegende Positionen im Text anspringen möchte, an denen man gleichzeitig arbeitet (z.B. ich an diesem Handbuch).
Eine Möglichkeit um dies zu erreichen wäre, die entsprechenden Zeilen auszuzeichnen und dann mit den Cursor-Tasten anzuspringen, dies hat jedoch den Nachteil, daß man unter Umständen mehrere Bewegungen im Text machen muß, um die gewünschte Stelle zu erreichen.
In diesem Fall gibt es aber eine weitere Besonderheit von Edward, die sogenannten „Lesezeichen". Mit der linken ALT-Taste und einer Zahlentaste des alphanumerischen Teils der Tastatur können Sie eine Zeile zwischenmerken. Wenn Sie nun an eine andere Stelle des Textes fahren und die linke AMIGA-Taste in Verbindung mit derselben Zahlentaste drücken, so springen Sie augenblicklich an die alte Stelle im Text.
Mit der Standard-Tastaturbelegung Edwards können Sie zehn Lesezeichen benutzen, theoretisch erlaubt Ihnen Edward jedoch 65536 Lesezeichen pro Text.

- Laden Sie eine Datei ein.
- Drücken Sie die linke ALT-Taste und die „1".

Es erscheint in der Titelzeile des Fensters die Meldung „Markierung gesetzt".

- Fahren Sie an eine beliebige andere Stelle des Textes.
- Drücken Sie nun die linke AMIGA-Taste und wieder die „1". Und schwupps sind Sie wieder an der alten Position im Text.

IX. Das Edit-Prefs Formular - Ändern der Voreinstellungen für die Texteingabe

Mit dem „Edit Prefs"-Formular, das Sie durch Wahl von „Edit Prefs..." aus dem „Text"-Menü erhalten, können Sie für jeden Text den Sie mit Edward bearbeiten diverse Einstellungen vornehmen.

1) Ist 1) aktiviert, dann wird bei der Standard-Tastaturbelegung auch nach Drücken der RETURN-Taste die neue Zeile entsprechend der alten eingerückt, wie es nach Drücken der ENTER-Taste der Fall wäre.

☞ Siehe auch Kapitel III. „Eingabe und Bearbeitung von Texten"

2) Durch Klick auf 2) schalten Sie den „Word-Wrap"-Modus an oder ab.

☞ Über den „Word-Wrap"-Modus erfahren Sie mehr im Kapitel X. „Word-Wrap".

3) Ist Overstrike angewählt, dann überschreiben Sie jeweils das Zeichen auf dem der Cursor steht durch das neugetippte. Normalerweise würde Edward das Zeichen einfügen, Overstrike überschreibt dagegen die alten Zeichen.

4) Ist diese Option aktiv, so werden alle eingegeben TABs in einer Zeile automatisch in entsprechend viele Leerzeichen umgewandelt.

5),6),7) Mit diesen drei Symbolen können Sie angeben, ob Tabs, Spaces und Zeilenenden einfach durch Leerräume dargestellt werden sollen (Normaleinstellung) oder besonders gekennzeichnet dargestellt werden sollen.

8) Dateien von anderen Computersystemen enthalten unter Umständen ein zusätzliches Zeichen um ein Zeilenende anzuzeigen. Durch Aktivieren dieser Option wird ein solches Zeichen automatisch beim Laden des Textes entfernt.

9) Tab Width gibt die Anzahl von Zeichen zwischen zwei Tabularpositionen an. Sie können diesen Abstand im Bereich 1 bis 20 wählen.

10) Mit Right Margin geben Sie die Position in einer Zeile an, ab der im „Word-Wrap"-Modus umgebrochen werden soll.

12) Wählen Sie USE, so werden die von Ihnen gewählten Einstellungen für den Text übernommen. Falls Sie „Right Margin" geändert haben, so wird der komplette Text dementsprechend neu umformatiert und der Cursor danach auf die erste Zeile des Textes gesetzt.

13) Verwirft die von Ihnen vorgenommenen Einstellungen.

X. „Word-Wrap“

Zum Erstellen von Fließtexten, wie sie unter anderem für DTP-Programme benötigt werden, können Sie Edward in dem „Word-Wrap“-Modus versetzen. „Word-Wrap“ bezeichnet ein spezielles Verhalten des Editors, bei dem eingegebene Zeilen beim Überschreiten einer bestimmten Länge umgebrochen werden. Ein Wort, das mitten auf der Umbruchposition liegt, wird automatisch in die nächste Zeile gezogen. Wird diese Zeile dadurch ebenfalls zu lang, so wird das Ganze fortgesetzt und zwar bis an das Ende eines Absatzes.

Ein Absatz wird dadurch gekennzeichnet, daß Sie nach der letzten Zeichenfolge des Absatzes „RETURN“ drücken. Im Gegensatz zum normalen Editier-Modus von Edward ist es im „Word-Wrap“-Modus nicht notwendig, die RETURN-Taste zu drücken, um eine neue Zeile zu erhalten. Vielmehr erzeugt Edward automatisch beim Umbruch neue Zeilen. Sie brauchen also nur draufloszutippen. Daher stammt auch die Bezeichnung „Fließtext“.

☞ Die Umbruchposition wird durch Angabe des rechten Randes im „Edit Prefs“-Formular angegeben. Der „Word-Wrap“-Modus kann ebenfalls durch dieses Formular ein- und ausgeschaltet werden. Wie Sie dabei vorgehen müssen, haben Sie im Kapitel IX. „Das Edit-Prefs Formular“ gelesen.

Im Gegensatz zu anderen Editoren, die einen Absatz erst durch Wahl eines bestimmten Menüpunktes einmalig umbrechen können, formatiert Edward den Text schon während der Eingabe. Auch beim Löschen innerhalb eines Absatzes werden Worte aus folgenden Zeilen eines Absatzes wieder nachgezogen, wenn in der aktuellen Zeile durch das Löschen Platz für sie gewonnen wurde.

Neben dem automatischen Umbruch des „Word-Wrap“-Modus können Sie einen Text auch nachträglich umformatieren.

Hierzu wählen Sie entweder „Format Paragraph“, um einen Absatz neu umzubrechen oder „Format Text“, um den gesamten Text neu umzubrechen. Beide Menüpunkte befinden sich im „Extras“-Menü.

XI. Makros

1) Makros aufzeichnen & abspielen

2) Makros speichern & abspielen

Eine Besonderheit von Edward, die Ihnen viel Arbeit erleichtern kann, ergibt sich aus der Möglichkeit, Makros aufzeichnen und abspielen zu können.
Fast alle Aktionen, die Sie beim Editieren eines Textes ausführen, können Sie wie mit einem Kassettenrekorder aufnehmen und bei Bedarf wiedergeben lassen. Das bedeutet, daß Sie sich alle gleichförmigen langweiligen Vorgänge, die Sie oft wiederholen müßten, durch Makros abnehmen lassen können.

1) Makros aufzeichnen & abspielen

Wenn Sie ein Makro aufzeichnen wollen, so wählen Sie den Menüpunkt „Aufnehmen" aus dem „Macro"-Menü. Daraufhin werden alle folgenden Aktionen, die Sie vornehmen, aufgezeichnet. Mit „Anhalten" aus dem „Macro"-Menü beenden Sie eine Aufzeichnung. „Abspielen" spielt das Makro dann für Sie ab.

- Laden Sie einen größeren Text ein.

Angenommen Sie wollten nun zu Beginn jeder Zeile die Zeichenfolge „Start>" und am Ende jeder Zeile die Zeichenfolge „<End" einfügen. Normalerweise müßten Sie jedesmal an den Anfang einer Zeile gehen, „Start>" eintippen, dann ans Ende einer Zeile springen, „<End" eingeben und in die nächste Zeile gehen. Wenn Sie die Möglichkeiten der Makros ausnutzen, wird das Ganze für Sie sehr viel einfacher.

- Fahren Sie mit dem Cursor in die erste Zeile Ihres Textes.
- Wählen Sie nun den Menüpunkt „Aufzeichnen" aus dem „Macro"-Menü.

Alle folgenden Aktionen werden nun aufgezeichnet.

- Drücken Sie SHIFT-CursorLeft. Damit springen Sie an den Anfang einer Zeile.
- Tippen Sie nun „Start >" ein.

- Anschliessend bewegen Sie den Cursor mit SHIFT-CursorRight an das Ende der aktuellen Zeile.

- Geben Sie nun „< End“ ein.

- Fahren Sie mit dem Cursor eine Zeile tiefer.

- Wählen Sie „Anhalten“ aus dem „Macro“-Menü. Damit haben Sie die Aufzeichnung beendet. Alle Aktionen zwischen „Aufnehmen“ und „Anhalten“ wurden aufgezeichnet und stehen nun zu Ihrer Verfügung.

Sie können nun jederzeit das gerade aufgezeichnete Makro abspielen.

- Wählen Sie „Abspielen“ aus dem „Macro“-Menü. Sie sehen, wie die folgende Zeile in Sekundenbruchteilen ebenfalls mit „Start >“ und „< End“ eingerahmt wird. Sie können diesen Vorgang nun beliebig oft wiederholen und sich damit viel Tipparbeit ersparen.

Statt obigem zugegebenermaßen etwas sinnlosem Beispiel werden Ihnen in der Praxis sicherlich sehr viel nützlichere Einsatzmöglichkeiten für die Makrofunktionen einfallen.

2) Makros speichern & abspielen

Die mit „Aufnehmen“ aufgezeichneten Makros stehen hier nur kurze Zeit zur Verfügung. Wenn Sie ein neues Makro aufzeichnen oder Edward verlassen, so ist Ihr zuletzt aufgezeichnetes Makro für immer verloren. Damit Sie ein besonders praktisches Makro nicht immer wieder neu aufzeichnen müssen, bietet Ihnen Edward die Möglichkeit ein Makro abzuspeichern und abgespeicherte Makros erneut abzuspielen.

- Falls Sie das Beispiel aus Abschnitt 1) noch nicht durchgespielt haben, so sollten Sie es jetzt tun. Wir benötigen für die folgenden Schritte nämlich ein frisch aufgezeichnetes Makro.

- Wählen Sie nun den Menüpunkt „Speichern als ...“ aus dem „Macro“-Menü. Es erscheint der Dateiauswahlfenster. Speichern Sie das Makro nun unter irgendeinem Namen ab.

- Zeichnen Sie ein anderes Makro auf oder verlassen Sie Edward und starten ihn danach erneut. Dies dient dazu das aktuelle Makro zu „vergessen“. Am einfachsten

ist dies für Sie, wenn Sie irgendwelche neuen Aktionen wie in Abschnitt 1) beschrieben aufnehmen.

- Wählen Sie nun „Open & Play“ aus dem „Macros“-Menü. Mit dem Dateiauswahlfenster geben Sie nun das im vorletzten Schritt abgespeicherte Makro an.

Sie werden sehen, wie Edward automatisch die aktuelle Zeile mit „Start >“ und „< End“ umrahmt.

Somit können Sie auch ein gespeichertes Makro beliebig oft wiederaufrufen.

☛ Ein mit „Öffnen und abspielen“ abgespieltes Makro läßt sich jederzeit mit „Abspielen“ aus dem „Macro“-Menü aufrufen, ohne erneut geladen werden zu müssen. Ein altes von Ihnen aufgezeichnetes Makro wird durch „Öffnen und abspielen“ überspielt.

☞ Im Zusammenhang mit „ARexx“ ergeben sich einige weitere Möglichkeiten. Lesen Sie dazu bitte das Kapitel XXI. „Edward - Kommunikation mit ARexx“.

XII. Fenster

1) Erzeugen von Fenstern

2) Wahl eines bestimmen Fensters

3) Schließen von Fenstern

4) Arbeiten mit mehreren Ansichten eines Textes

Sie wissen jetzt schon, wie man verschiedene Texte bearbeitet. Vielleicht haben Sie sich auch schon gefragt, wie man verschiedene Texte oder mehrere Ausschnitte eines bestimmten Textes gleichzeitig anzeigen kann.
Edward ermöglich dies durch die sogenannten Fenster. Was ein Fenster auf der Workbench oder im CLI bedeutet ist klar. Bei Edward ist ein Fenster ein normales Amiga-Fenster verbunden mit den Informationen, die Edward benötigt, um darin einen bestimmten Text oder Textausschnitt anzuzeigen. Wie das genau funktioniert, erkläre ich Ihnen in den folgenden Abschnitten.

1) Erzeugen von Fenstern

Nach dem Start öffnet Edward sofort ein Fenster, in welchem Sie einen Text bearbeiten können.

- Geben Sie irgendeinen Text ein.

- Wählen Sie aus dem „Fenster"-Menü den Eintrag „Neu". Sie sehen kurz, wie sich ein Fenster öffnet. Dieses Fenster liegt genau über dem ersten Fenster von Edward und hat auch denselben Inhalt. Wenn Sie nur flüchtig zugeschaut haben, werden Sie keinen Unterschied in der Anzeige bemerkt haben.

- In der rechten unteren Ecke befindet sich ein Symbol mit welchem Sie die Größe des Fenster verändern können. Halbieren Sie das Fenster in der Höhe und der Breite. Sie sehen nun, daß darunter Ihr altes Fenster liegt.

- Verschieben Sie das Fenster in die rechte, untere Ecke.

- Wählen Sie den Eintrag „Neu“ aus dem „Text“-Menü.

- Laden Sie einen Text mit „Projekt/Öffnen“, oder geben Sie einen weiteren Text ein.

Für den weiteren Verlauf unserer Übung sollten Sie jetzt noch ein Fenster öffnen, es verkleinern und in die linken unteren Ecke verschieben. Danach laden Sie einen weiteren Text oder Sie geben einfach ein paar Zeilen ein.

2) Wahl eines bestimmten Fensters

Es gibt verschiedene Möglichkeiten, ein bestimmtes Fenster zu aktivieren.

- Am einfachsten fahren Sie mit dem Mauspfeil im entsprechenden Fenster an die Position im Text, die Sie bearbeiten wollen und drücken die linke Maustaste. Das Fenster wird daraufhin aktiviert und der Cursor erscheint unter dem Mauspfeil. Über das Tiefen-Gadget (bzw. bei Kickstart 1.2, 1.3 die beiden Tiefengadgets) können Sie an jedes Fenster herankommen und es dann wie oben beschrieben mit dem Mauspfeil aktivieren.

- Wählen Sie auf diese Weise das Fenster in der rechten unteren Ecke an.

Ebenso wie die Texte werden auch die Fenster in einer Liste verwaltet. Auf einfache Weise können Sie das Fenster an der Listenposition vor oder auch nach dem momentan aktuellen wählen. Hierbei gilt ebenfalls alles zu den Texten gesagte. Das momentan aktuelle Fenster sollte das von Ihnen erzeugte in der linken unteren Ecke sein.

- Wählen Sie aus dem „Fenster“-Menü den Eintrag „Vorheriges“. Damit haben Sie wieder das Fenster in der rechten, unteren Ecke aktiviert.

- Wählen Sie erneut „Vorheriges“. Die beiden kleinen Fenster sind nun plötzlich verschwunden. Einzig das grosse Fenster, das beim Start von Edward automatisch geöffnet wurde, ist nun sichtbar. Das liegt daran, daß bei einer Aktivierung eines Fensters durch „Vorheriges“ oder „Nächstes“ dieses immer automatisch in der Vordergrund geholt wird.

3) Schließen von Fenstern

Sie können ein Fenster auf zwei Arten schließen. Entweder Sie bewegen dem Mauspfeil auf das Schließen-Symbol in der linken, oberen Ecke des Fensters und drücken die linke Maustaste, oder Sie wählen den Eintrag „Schließen" aus dem „Fenster"-Menü, der das momentan aktive Fenster schließt.

- Aktivieren Sie, wie im vorigen Abschnitt beschrieben, das große Fenster.
- Schließen Sie es nun durch den „Schließen"-Menüeintrag. Nun sind nur noch die beiden kleinen Fenster vorhanden.
- Schließen Sie nun das Fenster in der rechten, unteren Ecke mit dem Mauspfeil.

Wenn Sie nun mehrfach „Nächster" oder „Vorheriger" aus dem „Text"-Menü (nicht dem „Fenster"-Menü !) wählen, dann stellen Sie fest, daß die drei verschiedenen Texte alle noch da sind, obwohl Sie zwei Fenster geschlossen haben. Das liegt daran, daß beim Schließen eines Fensters nur die Anzeige eines Textes beendet wird. Der Text wird dadurch aber nicht freigegeben.

☛ Wichtig:
Wenn Sie versuchen, das letzte Fenster von Edward zu schließen, geht Edward davon aus, daß Sie die Arbeit beenden wollen. Sollten Sie noch Texte haben, deren Änderungen nicht abgespeichert wurden, so werden Sie durch ein Dialog-Fenster darauf aufmerksam gemacht. Gegebenenfalls können Sie die Operation mit „Nein" abbrechen. Wählen Sie „Ja", dann gehen alle vorher gemachten Änderungen verloren.

- Lesen Sie hierzu auch Abschnitt 3) aus Kapitel IV „Text".

4) Arbeiten mit mehreren Ansichten eines Textes

Mit Edward können Sie sich mehrere Ansichten desselben Textes anzeigen lassen und damit arbeiten.

- Sie sollten nun nur noch ein Fenster von Edward geöffnet haben. Sollte dies nicht der Fall sein, so sorgen Sie bitte mit den oben beschriebenen Schritten dafür.
- Bewegen Sie nun das Fenster in die linke, obere Ecke des Bildschirms und verändern Sie seine Größe so, daß es die gesamte Breite und etwa ein Drittel der Höhe des Bildschirms ausfüllt.

Damit Sie die in diesem Abschnitt beschriebenen Möglichkeiten richtig kennenlernen können, sollten Sie nun einen Text einladen oder eingeben, der etwa 100 Zeilen oder mehr umfasst.

- Erzeugen Sie zwei weitere Fenster mit denselben oben beschriebenen Ausmassen und ordnen Sie diese untereinander so an, daß es keine Überlappungen zwischen ihnen gibt.

- Aktivieren Sie nun das mittlere Fenster und fahren Sie mit dem Cursor nach unten in die 40zigste Zeile.

- Aktivieren Sie das unterste Fenster und fahren Sie mit dem Cursor in die letzte Zeile des Textes.

Sie können nun in jeder der drei Ansichten des Textes an verschiedenen Positionen weiterarbeiten. Der Vorteil ist, daß Sie in den beiden anderen Fenstern eine Übersicht über ganz andere Textstellen haben, als jene, die Sie gerade bearbeiten. Das kann besonders beim Erstellen von Quellcode für eine bestimmte Programmiersprache von Vorteil sein.

☛ Hinweis:
Alle Änderungen, die Sie an einem Text vornehmen, inklusive der Löschung desselben, werden nur in dem aktuellen Fenster auch angezeigt. Erst wenn Sie eine andere Ansicht des Textes in einem anderen Fenster anwählen, wird auch dort die Anzeige aktualisiert. Lassen Sie sich davon nicht verwirren.

Um Ihnen das Arbeiten mit verschiedenen Ansichten eines Textes zu erleichtern, bietet Edward zusätzliche Funktionen an.

- Schließen Sie alle Fenster bis auf eines.
- Wählen Sie nun „volle Größe“ aus dem „Fenster“-Menü.

Sie sehen, daß das letzte Fenster sich wieder auf die maximale Größe ausdehnt. So erhalten Sie schnell wieder den Überblick, falls ein Fenster zu klein war oder Sie durch Schließen anderer Fenster mehr Platz gewonnen haben.

- Wählen Sie nun „horizontal teilen“ aus dem „Fenster“-Menü. Daraufhin teilt sich das aktuelle Fenster horizontal in zwei gleichgroße Fenster auf. So erhalten Sie schnell die nötigen Fenster für verschiedene Ansichten desselben Textes. Mit „vertikal teilen“ unterteilen Sie genauso ein Fenster vertikal in zwei gleichgroße Fenster.

XIII. Verlassen des Editors

1) Unterbrechen der Arbeit

2) Verlassen des Editors

1) Unterbrechen der Arbeit

Edward bietet Ihnen besonderen Komfort, falls Sie die aktuelle Arbeit an einem oder mehreren Texten unterbrechen und den Editor verlassen wollen. Wenn Sie im „Projek t"-Menü den Eintrag „Projekt & Ende" wählen, so speichert Edward zusätzliche Informationen über alle momentan verwendeten Fenster und Texte ab. Wenn Sie Edward erneut starten, dann öffnet er automatisch Fenster und lädt Texte ein, so daß Sie die Arbeit in derselben Umgebung fortsetzen können, in der Sie sie unterbrochen haben.

☛ Hinweis:
Falls Sie Änderungen an Texten vornahmen, ohne diese zu speichern und daraufhin „Projekt & Ende" gewählt haben, so werden Sie über ein Dialog-Fenster gefragt, ob sie diese Texte „vergessen" wollen. Antworten Sie hier mit „Nein", dann wird die Operation abgebrochen. Wählen Sie „Ja", dann gehen die letzten Änderungen verloren!

- Damit Sie einen kleinen Eindruck über die Möglichkeiten der Arbeitsunterbrechung erhalten, sollten Sie nun Fenster mit verschiedenen Texten erzeugen.

☞ Hierzu können Sie noch einmal in den Kapiteln III. „Eingabe und ..." und XII. „Fenster" nachschauen

- Wählen Sie nun den Menüpunkt „Projekt & Ende" aus dem „Projekt"-Menü. Sie befinden sich nun wieder auf der Workbench oder im CLI, je nachdem, wie Sie Edward gestartet haben.

- Starten Sie Edward erneut. Automatisch öffnen sich die Fenster wieder so, wie Sie vor dem Verlassen angeordnet waren und zeigen die zuletzt bearbeiteten Texte an. Sie können nun einfach mit der Arbeit fortfahren.

2) Verlassen des Editors

Im Gegensatz zu „Projekt & Ende“ verlassen Sie mit „Ende“ aus dem „Projekt“-Menü den Editor, ohne daß Informationen zur momentanen Arbeitsumgebung gespeichert werden.

☛ Hinweis:
Falls Sie Änderungen an Texten vornahmen, ohne diese zu speichern und daraufhin „Ende“ gewählt haben, so werden Sie über ein Dialog-Fenster gefragt, ob sie diese Texte „vergessen“ wollen. Antworten Sie hier mit „Nein“, dann wird die Operation abgebrochen. Wählen Sie „Ja“, gehen die letzten Änderungen verloren!

XIV. Iconify & HotKey

1) Iconify

2) HotKey

1) Iconify

Benutzen Sie Edward auf dem Workbench-Screen, so wird es manchmal sehr lästig sein, wenn Edward mit seinen Fenstern etwas im Hintergrund verdeckt. Vielleicht wollen Sie einfach ein Programm von der Workbench aus starten oder in der Shell den Compiler anwerfen. Damit Sie sich nicht durch alle Fenster Edwards durchklicken müssen, gibt es die Funktion Ikonifizieren
Wählen Sie einfach aus dem „Fenster"-Menü den Eintrag „Ikonifizieren". Schon verschwinden alle Fenster, und es erscheint links oben auf dem Bildschirm ein Fenster, das nur aus einer Titelzeile besteht. In dieser steht „Iconified Edward". Edward ist nun nicht mehr aktiv. Das kleine Fenster könnnen Sie irgendwohin schieben, wo es Sie nicht stören kann.
Wollen Sie Edward wieder aktivieren, so haben Sie zwei Möglichkeiten. Die zweite wird im Abschnitt HotKey beschrieben.
Die erste Möglichkeit besteht darin, das Fenster mit dem Titel „Iconified Edward" zu aktivieren und dann die rechte Maustaste zu drücken. Daraufhin baut Edward wieder alle seine Fenster auf und Sie können weiterarbeiten.

2) HotKey

Um Edward nach einem Iconify wieder aktivieren zu können, oder um einfach das Fenster mit dem zuletzt bearbeiteten Text in den Vordergrund zu bekommen, gibt es den HotKey.
Der HotKey ist eine Taste in Verbindung mit bestimmten Qualifiern (SHIFT, ALT, CTRL, LEFT_AMIGA). Wird diese Taste gedrückt, so wird das Fenster mit dem zuletzt bearbeiteten Text und der Bildschirm auf dem sich dieses Fenster befindet in den Vordergrund gebracht. War Edward im Iconify-Zustand, so wird dieser aufgehoben, und alle Fenster werden wieder geöffnet.

☞ Näheres zur Definition des HotKeys erfahren Sie im Kapitel XX. „Benutzer-definierte Menüs, Tastaturbelegung und HotKey".

XV. Der Kommandozeilen-Interpreter

Die Bedienung Edwards über Menüs und Formulare stellt für den Anfänger sicherlich eine große Erleichterung dar, der Profi ist in vielen Fällen jedoch bereit auf diesen Komfort zu verzichten, wenn er nur besonders schnell die wichtigsten Funktionen benutzen kann.
Dies kann zum einen durch Definition von „Short-Cuts" wichtiger Menüpunkte und durch Belegung von Funktionstasten etc. geschehen. Eine weitere mächtige Möglichkeit ist der Kommandozeilen-Interpreter Edwards. Hier können viele Befehle und Befehlskombinationen Edwards direkt eingegeben und ausgeführt werden.

Die Eingabe einer Kommandozeile leiten Sie durch die „ESC"-Taste ein. Daraufhin erscheint der Cursor in der untersten Zeiledes aktiven Textfensters. Die dort angezeigte Textzeile wird durch eine Leerzeile ersetzt, in der Sie nun Ihre Eingaben machen können. Ein erneutes „ESC" bricht die Eingabe ab. Drücken Sie stattdessen „RETURN" oder „ENTER" werden die von Ihnen eingebenen Kommandos ausgewertet.
Mit „CTRL-O" gelangen Sie ebenfalls in den Eingabemodus des Kommando-Interpreters mit der Besonderheit, daß Ihre letzte Eingabe wieder zur Verfügung steht und geändert werden kann.
„CTRL-G" bewirkt eine erneute Ausführung der zuletzt eingebenen Kommandos.

☞ Weitere Informationen über den syntaktisch korrekten Aufbau einer Kommandofolge erhalten Sie in Kapitel XX. „Benutzerdefinierte Menüs, Tastaturbelegung und HotKey" in den Abschnitten 1) „Kurze Einführung in die EBNF-Darstellung" und 2) „Definition eigener Menüs", insbesondere aus der Definition der „CommandList".

☞ Alle verfügbaren Kommandos Edwards mit Kurzerläuterungen und Parameterbeschreibung sind in Anhang A) enthalten.

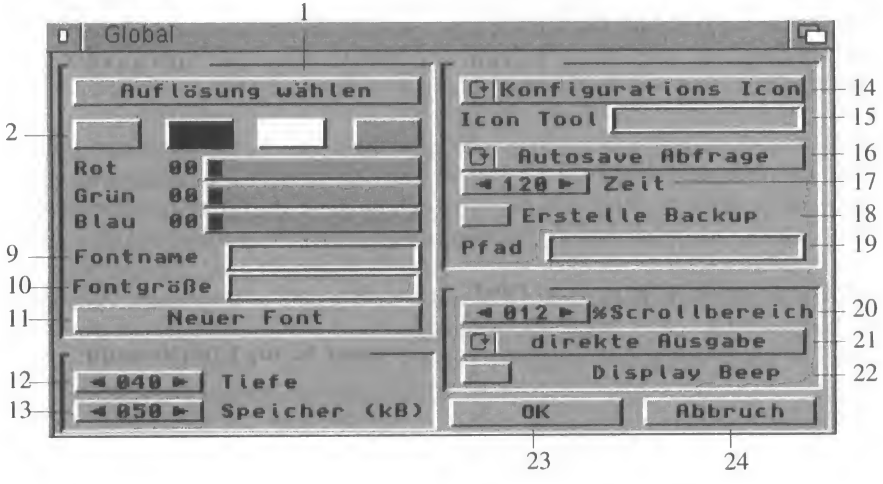
XVI. Globale Einstellungen

1) Das Formular für „globale Einstellungen“

2) Das „Seitenformat“-Formular

1) Das „Globale Einstellungen“-Formular

Mit dem „Globale Einstellungen“-Formular, das sich öffnet, wenn Sie „Einstellungen...“ aus dem „Projekt“-Menü wählen, können Sie verschiedene Einstellungen verändern, die den gesamten Editor betreffen.



1) Bei Klick auf dieses Symbol erscheint ein Dialog-Fenster aus dem Sie die gewünschte Auflösung auswählen können.

Mit den folgenden Symbolen können Sie die vier Farben einstellen, in denen die Fenster und Texte Edwards auf einem eigenen Bildschirm dargestellt werden sollen.

2) - 5) Durch Klicken auf eines dieser Symbole bestimmen Sie die Bildschirmfarbe, die verändert werden soll.

6) - 8) Mit diesen Reglern bestimmen Sie den Rot-, Grün- und Blauanteil der von Ihnen gewählten Farbe. Als kleine Hilfe wird rechts von den Reglern der Wert des jeweiligen Farbanteils zwischen 0 und 15 angezeigt.

9) In dieses Texteingabefeld können Sie den Namen des Zeichensatzes angeben, in welchem die Texte Edwards dargestellt werden sollen.

10) Dieses Texteingabefeld erfaßt die Fontgröße.

☛ Wichtig: Für den beschleunigten Bildaufbau bei der Textdarstellung mußten bei der Entwicklung Edwards gewisse Kompromisse eingegangen werden. Daher ist es leider nicht möglich Proportional-Zeichensätze zu wählen. Sollten Sie einen falschen Zeichensatz gewählt haben, so erfolgt keine Änderung der Textdarstellung.

11) Durch Anklicken dieses Symbols öffnet sich ein Dialog-Fenster, mit dem Sie auf komfortable Weise den gewünschten Zeichensatz wählen können.

12), 13) Mit diesen beiden Zahlenfeldern kontrollieren Sie das Verhalten von Edward im Zusammenhang mit der Undo-Funktion. Mit Undo-Depth geben Sie an, wie viele Operationen, die einen Text verändern, gemerkt werden sollen. Sollte Sie an einem Text eine Anzahl von Änderungen vornehmen, die diesen Wert überschreitet, so werden alte Änderungen „vergessen". Einfache Operationen, wie „Zeichen löschen" oder „Neues Zeichen einfügen" benötigen keinen Zusatzspeicher zum Merken. Das Ausschneiden eines Blockes dagegen kann nur gemerkt werden, wenn genügend Speicherplatz bereitgestellt wurde, um den ausgeschnittenenen Block zusätzlich speichern zu können. Würden Sie nun sehr oft Blöcke ausschneiden und kopieren, so würde sehr viel Speicher in Anspruch genommen. Mit „UndoMem" begrenzen Sie den Speicherplatz, den die Undo-Funktion in Anspruch nimmt, auf einen Maximal-Wert. Wird dieser überschritten, so werden alte Operationen „vergessen", um genügend Speicher wieder freizugeben.

☛ Wichtig: Da die Undo-Funktion für jeden Text die notwendigen Informationen zum Rücknehmen einer Veränderung merken muß, gilt Undo-Mem auch als Wert für jeden Text. Das bedeutet, daß Sie für jeden Text, den Sie bearbeiten, im ungünstigsten Fall die Menge von „UndoMem" zusätzlich an Speicher beanspruchen.

Mit den anschließend beschriebenen Symbolen können Sie das Verhalten Edwards beim Speichern von Dateien angeben.

14) Mit diesem Blätter-Symbol geben Sie an, ob Edward beim Speichern einer Datei ein Pictogramm anlegen („Configuration Icon") oder darauf verzichten soll („No Icon").

☛ Wichtig: Nur wenn Sie die Option „Configuration Icon" gewählt haben, werden beim Sichern einer Datei wichtige Zusatzinformationen ebenfalls mitabgespeichert. Dazu gehören unter anderem die aktuelle Position in Ihrem Text, der „Word-Wrap"-Modus und - sehr wichtig - die Strukturierung des Textes durch ausgezeichnete Zeilen.

15) Wenn Sie ein Pictogramm zu jedem Text abspeichern lassen, so können Sie mit diesem Eingabefeld bestimmen, welches Programm beim Doppelklick auf das Pictogramm des gespeicherten Textes geladen werden soll. Im Normalfall sollte dies „Edward:Edward" sein.

16) Edward kann die von Ihnen bearbeiteten Texte nach einer bestimmten Zeitspanne abspeichern. Mit dem Blätter-Symbol haben Sie die Wahlmöglichkeit zwischen :

- „Auto-Save OFF", das bedeutet das automatische Abspeichern von Texten ist abgeschaltet.
- „Auto-Save", dann speichert Edward nach der in 17) angebenen Zeitspanne alle veränderten Texte automatisch ab.
- „Confirm Auto-Save", dann fragt Edward vor dem automatischen Speichern von Texten um Erlaubnis.

17) Mit diesem Einsteller können Sie die Zeit in Sekunden angeben, die verstreichen darf, bis Edward automatisch die geänderten Texte abspeichert.

18) Hiermit bestimmen Sie, ob beim Speichern einer Datei eine möglicherweise schon vorhandene Datei gleichen Namens in ein anderes Verzeichnis umkopiert und umbenannt werden soll.

19) In diesem Eingabefeld geben Sie ein spezielles Muster an. Dieses besteht aus dem Pfadnamen gefolgt von einer Zeichenkette, die ein „*" enthalten sein kann. Beim Anlegen eines Backups einer Datei wird diese in den angebenen Pfad umkopiert. Daraufhin wird sie nach der oben erwähnten Zeichenkette umbenannt wobei der „*" durch den alten Namen der Datei ersetzt wird.

20) Mit Scrollbereich ist der Bereich innerhalb eines Textfensters gemeint, in dem der Cursor das Scrollen des Textes bewirkt. Sie können den Abstand des Cursors zu den Fenster-Randkomponenten in Prozent angeben.

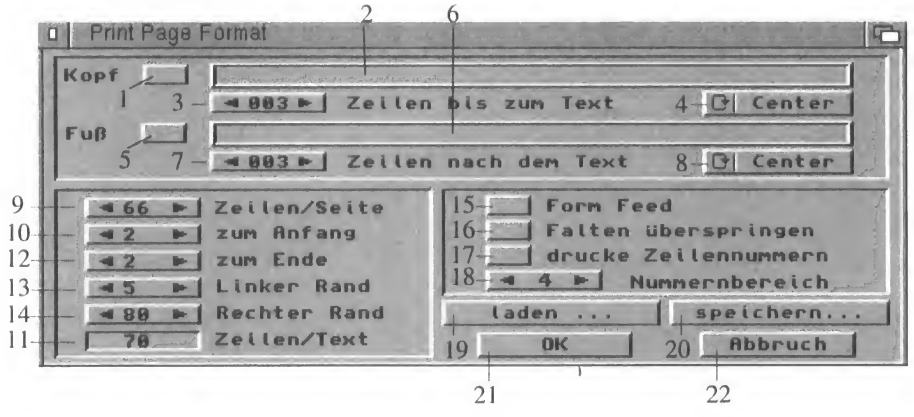
21) Durch das Blätter-Symbol können Sie einstellen, ob die Textausgabe mit Betriebssystem-Funktionen oder direkt gemacht werden soll. Diese Einstellmöglichkeit erhöht die Kompatibilität Edwards zu späteren Betriebssystem-Versionen und zu neuen Grafikkarten mit besonderen Fähigkeiten.

22) Mit diesem Symbol läßt sich der Display-Beep, der durch jeden Fehler erzeugt wird, abschalten.

23),24) Mit diesen beiden Symbolen übernehmen Sie die Einstellungen des Formulars oder Sie brechen den Vorgang ab.

2) Das „Seitenformat"-Formular

Mit dem „Seitenformat"-Menü können Sie bestimmen, wie ein Text auf dem Drucker ausgegeben werden soll.



1), 5) Mit diesen Gadgets geben Sie an ob eine Kopf- bzw. eine Fußzeile ausgedruckt werden soll.

2), 6) Hier wird der Text für die Kopf- bzw. Fußzeile angegeben. Spezielle Kürzel in diesem Text werden beim Druck interpretiert. Möglich sind hier: „%D" = Tag, „%M" = Monat, „%Y" = Jahr, „%h" = Stunde, „%m" = Minuten und „%s" = Sekunden. Durch diese Kürzel kann die aktuelle Zeit beim Ausdruck einer Seite in der Kopf- bzw. Fußzeile eingefügt werden.
Weiterhin kann durch „%p" die Seitennummer eingefügt werden.

3) Gibt die Anzahl Zeilen zwischen der Kopfzeile und dem Text an.

7) Analog zu 3) gibt 7) die Anzahl Zeilen zwischen Text und Fußzeile an.

4), 8) Hiermit läßt sich die Kopf- bzw. Fußzeile zentriert, links- oder rechtsbündig drucken.

9) Mit 9) geben Sie an, wieviele Zeilen inklusive Kopf- und Fußzeile, sowie aller Abstände auf ein Blatt passen.

10) Hiermit defininieren Sie den Abstand in Zeilen vom Blattbeginn zur ersten Zeile, die gedruckt werden soll (Dies kann die Kopfzeile oder die erste Textzeile sein).

11) Dient zur reinen Information. Sie sehen, wieviele Zeilen für die Ausgabe des Textes übrigbleiben.

12) Gibt analog zu 10) den Abstand in Zeilen zum Ende des Blattes an.

13) Der Abstand des linken Randes in Zeichen.
14) Die Breite einer Zeile in Zeichen.
15) Nach dem Drucken aller Seiten wird ein zusätzlicher FormFeed gesendet, wenn Sie diese Option aktivieren.
16) Hiermit geben Sie an, ob beim Druck eingefaltete Zeichen übersprungen werden sollen.
17) Wählen Sie diese Option an, so wird jede gedruckte Zeile durchnumeriert.
18) Die Stellenanzahl einer Zeilennummer kann hier angegeben werden.
19) Laden eines bestimmten Druckformates.
20) Speichern der Einstellungen.
21) Die Einstellungen werden übernommen.
22) Die Einstellungen werden verworfen.

XX. Benutzerdefinierte Menüs, Tastaturbelegung und HotKey

1) Kurze Einführung in die EBNF-Darstellung

2) Definition eigener Menüs

3) Definition der Tastaturbelegung

4) Definition des HotKeys

5) Einladen einer neuen Definition

In der Datei „Edward.DEF" befinden sich die Definitionen für die Menüs sowie die Tastaturbelegung Edwards. Diese sind in weiten Grenzen veränderbar. Der Aufbau der Definitionsdatei lehnt sich dabei an eine sehr primitive Programmiersprache an, besitzt also eine ganz einfache Syntax. Die folgenden Abschnitte werden Ihnen das näher erläutern.

1) Kurze Einführung in die EBNF-Darstellung

Tja, um Ihnen die Syntax der Definitionen ein bißchen näherzubringen muß ich jetzt leider etwas weiter ausholen und die Erweiterte-Backus-Naur-Darstellung einführen. Hinter diesem ominösen Namen, der jedem Informatik-Studenten mehr oder weniger bekannt ist, verbirgt sich eine Darstellungsform für die Syntax von Programmiersprachen. Mächtig kompliziert ist sie deswegen aber nicht.

Eine Definition beginnt immer mit dem Namen des zu Definierenden, daraufhin folgt ein „="-Zeichen. An dieses schließt sich die eigentliche Definition an. Sie gibt im wesentlichen an, welche Zeichenfolgen syntaktisch korrekt sind. Dies geschieht durch direkte Angabe von Zeichenfolgen oder der Angabe einer anderen Definition, in der dann nachgeschaut werden muß. Zeichenfolgen werden immer in Anführungsstrichen wiedergegeben. Jede Definition wird mit einem „." abgeschlossen.

Beispiel 1:

Wort = "Mahlzeit" .

Würde die Syntax unser Sprache nur aus obiger Definition bestehen, so wäre die einzige syntaktisch korrekte Zeichenfolge „Mahlzeit". Zugegebenermaßen etwas dürftig. Mir sind allerdings Zeitgenossen bekannt, deren Sprachschatz nicht signifikant umfangreicher ist.

Beispiel 2:

KlammerAusdruck = "(" Ausdruck ")".
Ausdruck = "Mahlzeit" .

In diesem Beispiel nimmt die Definition von Klammerausdruck bezug auf die Definition von Ausdruck. Der einzige korrekte Klammerausdruck ist "(Mahlzeit)".

Um anzuzeigen, daß mehrere Zeichenfolgen einer Definition genügen, dient folgende Notation : Name = (... | ...). wobei „..." für eine Zeichenfolge oder eine andere Definition steht.

Beispiel 3:

Wunsch = ("Guten Appetit"| KlammerAusdruck).
KlammerAusdruck = "(" Ausdruck ")" .
Ausdruck = "Mahlzeit" .

Ein Wunsch kann laut dieser Definition entweder aus der Zeichenfolge "Guten Appetit" oder der Zeichenfolge "(Mahlzeit)" bestehen.

Die bisher erwähnten Beschreibungsmöglichkeiten für die Syntax einer Sprache würden schon ausreichen, aber um Schreibarbeit zu ersparen, wurden noch einige Abkürzungsmöglichkeiten geschaffen.

Steht auf der rechten Seite einer Definition ein Ausdruck in eckigen Klammern, so bedeutet dies, daß der Inhalt der Klammern stehen kann oder auch nicht.

Beispiel 4:

Satz : "Jetzt geht es " ["endlich "] "zur Sache."

Die beiden korrekten Zeichenfolgen wären : "Jetzt geht es zur Sache." und "Jetzt geht es endlich zur Sache."

Eine letzte Schreibweise bedient sich der geschweiften Klammern. Ein Ausdruck in geschweiften Klammern kann beliebig oft oder auch gar nicht stehen.

Beispiel 5:

Satz : "Hallo " {"Echo "} "."

Korrekte Zeichenfolgen wären : "Hallo .", "Hallo Echo .", "Hallo Echo Echo ." usw..

Mit den obigen Erläuterungen sollten Sie nun den Aufbau der Definitionsdatei verstehen können. Falls Ihnen bisher alles zu theoretisch trocken vorkam, dann sollten Sie sich einfach mal die Definitionsdatei „Edward.DEFS" anschauen. Die Struktur, die die Definitionen haben müssen, sollte Ihnen auch durch einfaches Anschauen schnell klarwerden.

2) Definition eigener Menüs

Die Syntax der Menü-Definitionen ist wie folgt definiert.

```
Menu        = "MENU" MenuTitle { MenuGroup } .
MenuGroup   = "TITLE" QuotedString { MenuItem } .
MenuItem    = ("ITEM" QuotedString CommandList [QuotedString] |
               "EMPTY" ) .

CommandList = ( "(" Command ")" | "(" {CommandList} ")"  ) .
Command     = ( EdwardCommand | CommandList ) .
```

QuotedString ist eine Zeichenfolge zwischen zwei doppelten Anführungszeichen (z.B. "Projekt" für das Projekt-Menü).
EdwardCommand ist ein Kommando des Editors mit Argumenten (falls erforderlich), wie es im Anhang A) „Die Kommandos des Editors" beschrieben wird.

Zur Bedeutung der einzelnen Definitionen :

Menu: Die Menüdefinitionen in der Datei „Edward.DEF“ müssen mit dem Schlüsselwort „MENU“ beginnen. Daraufhin folgen die Definitionen der Einträge (MenuGroup), wobei mindestens ein Eintrag definiert werden muß.

MenuGroup: Eine Menü-Gruppe beginnt mit dem Schlüsselwort „TITLE“, dem sich ein QuotedString anschließt. Die Zeichenfolge des QuotedString (s.o.) gibt den Titel der Menü-Gruppe an, der erscheint, wenn Sie die rechte Maustaste drücken.
Auf die Angabe des QuotedSting können beliebig viele Einträge folgen.

MenuItem: Ein Eintrag in einer Menü-Gruppe (MenuGroup) besteht entweder aus dem Schlüssel-Wort „EMPTY“, dieses dient dazu verschieden Einträge in einer Gruppe voneinander durch einen Balken zu trennen, oder dem Schlüsselwort „ITEM“ und einigen weiteren Angaben. „ITEM“ muß in jedem Fall ein QuotedString folgen. Dieser gibt den Text des Menü-Eintrags an, der erscheint, wenn Sie mit der Maus die entsprechende Menü-Gruppe aufklappen lassen. Daran schließt sich eine CommandList an. Die CommandList ist eine Folge von Kommandos mit Argumenten, die Edward versteht. Diese werden jedesmal ausgeführt, wenn Sie den entsprechenden Eintrag in einer Menü-Gruppe mit der Maus angewählt haben.
Mit einem weiteren wahlweisen QuotedString können Sie ein Zeichen angeben, welches als Short-Cut dienen soll.

3) Definition der Tastaturbelegung

```
Keys          = "KEYS" { ModeList }
ModeQualList  = "("{ Mode | Qualifier | ""}{ KeyDef }")"
Mode          = ( "MODE1" | "MODE2" | "MODE3" )
Qualifier     = ( "ALT" | "SHIFT" | "CTRL" | "LEFT_COMMAND" )
KeyDef        = ( KeyName | QuotedString | "NP" QuotedString )
                CommandList .
```

```
KeyName = (
"F1" | "F2" | "F3" | "F4" | "F5" | "F6" | "F7" | "F8" | "F9" |
"F10" | "ESC" | "BACK" | "DEL" | "HELP" |
"CRSRUP" | "CRSRDOWN" | "CRSRRIGHT" | "CRSRLEFT" |
"TAB" | "RETURN" | "SPACE" | "ENTER" |
"NP_NUML" | "NP_SCRL" | "NP_PRTSC" | "NP_HOME" | "NP_UP" |
"NP_PGUP" | "NP_LEFT" | "NP_RIGHT" | "NP_END" | "NP_DOWN" |
"NP_PGDN" | "NP_INS" | "NP_DEL" )


CommandList = ( "(" Command ")" | "(" {CommandList} ")"  )
Command     = ( EdwardCommand | CommandList )
```

QuotedString ist eine Zeichenfolge zwischen zwei doppelten Anführungszeichen (z.B. „Project“ für das Project-Menü).
EdwardCommand ist ein Kommando des Editors mit Argumenten (falls erforderlich), wie es im Anhang A) „Die Kommandos des Editors“ beschrieben wird.

Wie ein Tastendruck interpretiert werden soll, hängt von verschiedenen Umständen ab. Wichtig ist zunächst einmal, ob die Taste in Kombination mit ALT, CTRL, SHIFT oder LEFT_COMMAND (der linken AMIGA-Taste) betätigt wurde. ALT, CTRL, SHIFT und LEFT_COMMAND werden auch als Qualifier bezeichnet. Für jede beliebige Kombination aus diesen Qualifiern wird der Tastendruck gesondert interpretiert. Das bedeutet, wenn Sie ALT-F1 drücken, geschieht etwas anderes, als wenn Sie ALT-CTRL-F1 drücken. Drücken Sie eine Taste, ohne gleichzeitig eine oder mehre Qualifier-Tasten gedrückt zu haben, so wird auch das gesondert interpretiert. Daraus ergeben sich sechzehn verschiedene Möglichkeiten eine Taste zu belegen (2 * 2 * 2 *2; (mit ALT und ohne ALT) * (mit CTRL und ohne CTRL) * (mit SHIFT und ohne SHIFT) * (mit LEFT_AMIGA und ohne LEFT_AMIGA)).

Das war schon ganz schön kompliziert, aber es wird noch viel schlimmer.
Mit den Befehlen „SetMode1“, „SetMode2“ und „SetMode3“ können Sie einen speziellen Modus setzen. Ist dieser Modus gesetzt, so wird eine daraufhin gedrückte Taste in Kombination mit allen Qualifiern wieder speziell interpretiert. Danach wird der Modus wieder zurückgesetzt.
Wurde eine Taste nicht mit einer CommandList belegt, so wird beim Druck der Taste das ihr entsprechende ASCII-Zeichen in den Text eingefügt. Das bedeutet für Sie eigentlich nur, daß Sie ein „a“ bekommen, wenn Sie die Taste mit dem „a“ drücken. Es sei denn, aus irgendwelchen Gründen hätten Sie diese Taste mit einer CommandList belegt.

Was das alles zu bedeuten hat, muß Ihnen jetzt noch nicht sonnenklar aufgegangen sein, zumindest können Sie jetzt einen Zusammenhang mit der oben angegebenen Syntax erkennen, den ich Ihnen aber gerne noch etwas näher erläutere.

Keys: Jede Definition der Tastaturbelegung muß mit dem Schlüsselwort „KEYS" beginnen. Diesem schließt sich eine beliebige Anzahl von ModeLists an.

ModeQualList: Bei diesem Wort sollte es jetzt bei Ihnen klingeln. In einer ModeQualList geben Sie an, wie eine Taste in Verbindung mit den gesetzten Modes und Qualifiern interpretiert werden soll. Nach der öffnenden runden Klammer „(„ mit der eine ModeQualList beginnt, folgt eine beliebige Kombination aus Mode- und Qualifier-Angaben (z.B.: „(MODE2 ALT SHIFT ..." oder gar nichts.
Es schliessen sich die KeyDefs an und zum Schluß kommt die obligatorische schliessende runde Klammer „)".

KeyDef: Eine KeyDef beginnt mit der Angabe der zu belegenden Taste. Ist dies eine Sondertaste, so geben Sie deren speziellen Namen an (alle Namen sind oben aufgeführt worden). Ansonsten müssen Sie in einem QuotedString das Zeichen angeben, das erscheint, wenn Sie die Taste ohne irgendeinen Qualifier drücken (also ohne SHIFT, ALT, CTRL). Befindet sich die Taste auf dem numerischen Ziffernblock, so geben Sie vor dem QuotedString das Schlüsselwort NP (NP für NumericPad = Zahlenblock) an.
Der Angabe der zu belegenden Taste folgt eine CommandList. Diese gibt die Kommandos an, die Edward nach dem Drücken der entsprechenden Taste ausführen soll.

Damit Sie nun nicht glauben, nur Nobelpreisträger könnten die Tastaturbelegung Edwards ändern, werde ich Ihnen nun am Ende des Abschnittes ein paar Beispiele vorführen.

Beispiel 1 : Zusammenhang zwischen Qualifiern und Taste

```
KEYS
(
    DELETE      (delete)
)
(ALT CTRL
    DELETE (insertchars "Auf 'nem PC wär das das Ende" )
)
```

Mit der obigen Definition wird beim Drücken von DELETE (Taste „DEL“) in Verbindung mit ALT und CTRL der Text „Auf 'nem PC wär das das Ende“ ab der aktuellen Cursor-Position eingefügt. Dieses Beispiel befindet sich schon in der Datei „Edward.DEFS“. Probieren Sie es ruhig mal aus. Drücken Sie dagegen nur DELETE, so wird das Zeichen rechts vom Cursor gelöscht.

Beispiel 2 : Zusammenhang zwischen Modi und Taste

```
KEYS
(CTRL
  "x"   (SetMode1)
)
(MODE1 CTRL
  "c" (insertchars  "EMACS Quit" )
  "s" (insertchars  "EMACS Save" )
)
```

Obige Defnition bewirkt folgendes :
Wenn Sie CTRL-x drücken wird Modus 1 eingestellt. Die nächste Taste wird also gemäß den Angaben für Modus 1 interpretiert. Danach wird automatisch wieder der Normal-Modus eingestellt. Wenn Sie also CTRL-x und gleich anschließend CTRL-C drücken, so wird er Text „EMACS Quit“ rechts vom Cursor eingefügt.
Auch dieses Beispiel ist schon in der Datei „Edward.DEF“ enthalten. Sie können es daher gleich ausprobieren.

Sollten immer noch nicht alle Klarheiten beseitigt sein, dann laden Sie die Datei „Edward.DEF“ einfach ein und schauen sich die Definitionen an, verändern Sie auch ruhig ´mal etwas, aber vergessen Sie nicht, sich eine Sicherheitskopie von „Edward.DEF„ anzulegen.

4) Definition des HotKeys

```
HotKey     = "HOTKEY" Qualifier HotKeyDef
Qualifier  = ( Qualifier | "ALT" | "SHIFT" | "CTRL" |
             "LEFT_COMMAND" | "" ) .
HotKeyDef  = ( KeyName | QuotedString | "NP" QuotedString ) .
```

```
KeyNam = (
"F1" | "F2" | "F3" | "F4" | "F5" | "F6" | "F7" | "F8" | "F9" |
"F10" | "ESC" | "BACK" | "DEL" | "HELP" |
"CRSRUP" | "CRSRDOWN" | "CRSRRIGHT" | "CRSRLEFT" |
"TAB" | "RETURN" | "SPACE" | "ENTER" |
"NP_NUML" | "NP_SCRL" | "NP_PRCSC" | "NP_HOME" | "NP_UP" |
"NP_PGUP" | "NP_LEFT" | "NP_RIGHT" | "NP_END" | "NP_DOWN" |
"NP_PGDN" | "NP_INS" "NP_DEL" ).
```

Wie Sie im Kapitel „XIV. Iconify & HotKey" nachlesen konnten, kann Edward nach Ausführen der Iconify-Funktion wieder durch einen HotKey aktiviert werden. Welche Taste als HotKey dient, wird ebenfalls in der Datei „Edward.DEF" angegeben. Die Syntax hierfür ist sehr einfach. Die Definition des HotKeys beginnt mit dem Schlüsselwort „HOTKEY". Es folgt eine Liste von Qualifiern wie z.B. ALT oder LEFT_AMIGA und schließlich wird die Taste angegeben, die hinfort die Rolle des HotKeys spielen soll.

Beispiel:

```
HOTKEY
  CTRL ALT DELETE
```

Das bedeutet, wenn Sie Edward in den Iconify-Zustand geschickt haben, so können Sie Ihn durch Drücken von CTRL, ALT und der DELETE-Taste wieder aktivieren. Auf einem PC würde diese Tastenkombination übrigens einen Soft-Reset ausführen.

5) Einladen einer neuen Definition

Normalerweise lädt Edward die Definition der Menüs, der Tastatur, etc. gleich beim Start aus der Datei „Edward.DEF". Sie können aber auch später die einzelnen Definitionen ändern, indem Sie eine andere Definitionsdatei einladen. Dies ist durch Wahl von „Open Defs" aus dem „Prefs"-Menü möglich. Es erscheint dann ein Dateiauswahlfenster, mit dem Sie eine andere Definitionsdatei angeben können.

XXI. ARexx - Kommunikation mit Edward

Diese Kapitel kann leider (für Sie) und zum Glück (für mich, den Autor dieses Handbuches) keine Einführung in die Programmiersprache Rexx sein. Dies würde den Rahmen dieses Handbuches bei weitem sprengen.
Außerdem gibt es inzwischen genügend Sekundärliteratur zu diesem Thema. Was soll dann dieses Kapitel, werden Sie sich fragen?
Ganz einfach: Ihnen die Möglichkeiten im Zusammenspiel zwischen Edward und ARexx aufzeigen.
ARexx wurde ja gerade dazu geschaffen, die unterschiedlichsten Programme von außen zu steuern und in ihren Ablauf einzugreifen.
Edward hat auch auf diesem Gebiet eine ganze Menge zu bieten. Das Meiste verbirgt sich jedoch unter der Oberfläche.
Die einzige Funktion, die Sie direkt über Menüs ansprechen können, ist der Menüpunkt „ARexx-Befehl“ aus dem „Macro“-Menü. Wenn Sie diese Funktion wählen, so erscheint ein Dateiauswahlfenster, mit welchem Sie ein ARexx-Programm auswählen können. Dieses wird daraufhin ausgeführt.
Alles Weitere, was sich aus dem Zusammenspiel zwischen ARexx und Edward ergibt, wird erst über die speziellen Kommandos Edwards möglich, mit denen intern alle Funktionen, die Sie z.B. über die Menüs aktivieren, ausgeführt werden. Alle diese Kommandos können auch über ARexx angesprochen werden. Dazu ist es notwendig, in Ihrem ARexx-Programm die folgende Zeile einzufügen :

```
ADDRESS "EDWARD"
```

Alle folgenden Befehle, die ARexx selbst nicht interpretieren kann, werden dann an Edward geschickt. Welche Befehle dies sein können, erfahren Sie im Anhang A) „Die Kommandos des Editors“. Die Befehle in diesem Anhang sind thematisch geordnet. Im Abschnitt XIV. des Anhangs A) finden Sie zudem spezielle Kommandos, die nur im Zusammenspiel mit ARexx sinnvoll sind.

XXII. Zurücknehmen von Text-Änderungen

Mit den Funktionen „Undo" und „Redo" erhalten Sie durch Edward die Retter in der Not, die Sie wahrscheinlich schon oft vermißt hatten. Wie oft hat man doch schon an einem Text etwas geändert, schnell mal was umkopiert oder gelöscht und dann festgestellt, daß das eigentlich nicht gerade die beste Idee war. Was dann? Natürlich könnte man den alten Text wieder einladen und von Neuem beginnen. Wie lästig.
Zum Glück gibt es die Undo-Funktion in Edward. Damit können Sie eine von Ihnen vorgegebene Anzahl aller Operationen, die einen Text verändert haben, wieder rückgängig machen.
Mit „ReDo" ist es Ihnen sogar möglich, die Rücknahme der Änderungen wieder zurückzunehmen (ganz schön komplex).
Natürlich macht Edward das Ganze nicht umsonst.
Um sich alle Änderungen merken zu können, braucht er Speicher. Wieviel, daß können Sie mit den „Global Prefs" ebenso wie die Anzahl der zu merkenden Operationen einstellen.

☞ Siehe hierzu insbesondere Kapitel XVI. Globale Einstellungen 1) Das Formular für globale Einstellungen.

Um einen kleine Eindruck der „UnDo"-Funktion zu erhalten, probieren Sie einfach das folgenden Beispiel aus.

- Geben Sie folgende Zeile ein: „Das Nichts als Negation des Seins mit dem Potential der Realisation der Existenz."
- Wählen Sie nun „UnDo" aus dem „Extras"-Menü.

Daraufhin verschwindet der letzte Buchstabe, den Sie getippt hatten.
Sie können dies wiederholen, bis irgendwann die Meldung „Everything is undone!" in der Titelzeile erscheint.

- Wählen Sie nun „ReDo" aus dem „Extras"-Menü. Daraufhin erscheint der Buchstabe wieder, der durch die vorige Undo-Operation gelöscht wurde.
- Auch „ReDo" können Sie wiederholen, bis wieder die gesamte Zeile dasteht.

Vielleicht laden Sie jetzt mal einen größeren Text ein, schneiden Blöcke aus und kopieren Sie um, löschen einige Zeilen und tippen neue Zeichen ein.
Wenden Sie dann die Undo-Funktion an.

☛ Hinweis:
Auch die Undo-Funktion stößt einmal an ihre Grenzen. Wenn sehr viel Speicher zum Merken von Textveränderungen benötigt wird, dann kann es vorkommen, daß Edward Speicher, der zum Merken älterer Veränderungen benötigt wurde, freigeben muß. Diese Änderungen sind dann nicht mehr rückgängig zu machen.

Weiterhin können nur so viele Änderungen gemerkt werden, wie Sie in den „Global Prefs“ als Undo-Tiefe angegeben haben. Kommen mehr Änderungen zusammen, so werden die ältesten Änderungen vergessen.

Anhang A) „Die Kommandos des Editors“

In diesem Teil des Anhangs erhalten Sie eine thematische Übersicht über alle verfügbaren Kommandos Edwards. Der Hinweis „Referenz“ zu Beginn eines Abschnitts gibt Ihnen das jeweilige Kapitel an, in welchem Sie weitere Informationen zu dem vorgestellten Kommando erhalten können.
Alle Kommandos werden in einem einheitlichen Format dargestellt. Bei der Angabe der Parameter gilt es folgendes zu beachten :
Die Parameter werden von p_0, p_1 bis p_n durchnumeriert. Hinter jedem Parameter wird sein Typ angegeben.

Beispiel:
p_0=STRING
bedeutet, daß der erste Parameter ein String sein soll.

Es gibt folgende Parametertypen :

1) VALUE: Eine ganze Zahl.
2) STRING: Eine durch Anführungszeichen „ begrenzte Zeichenfolge.
3) LIST: Eine durch „(“ und „)“ begrenzte Aufzählung von Elementen. Die Elemente dürfen dabei VALUES, STRINGS und natürlich auch LISTen sein. Im allgemeinen erwarten die Kommandos Edwards jedoch ihre Parameter in einer genau spezifizierten Reihenfolge. Informationen darüber sind dann aber in der Beschreibung des Befehls zu finden. Listen dienen ebenfalls der Rückgabe von Werten, insbesondere im Zusammenhang mit der ARexx-Schnittstelle.

Ein „OPT_“ vor einem Parameter bedeutet, daß er optional ist und auch weggelassen werden darf.

Als Rückgabewert ist außer VALUE, STRING und LIST zusätzlich noch der sogenannte SMALL_STRING möglich. Hierbei handelt es sich um eine Zeichenfolge, die nicht durch Anführungsstriche begrenzt wird. Dies ist allerings nur im Zusammenhang mit der ARexx-Schnittstelle sinnvoll, da die Kommandos Edwards ihre String-Parameter immer als STRING erwarten.

I. Eingabe und Bearbeitung von Texten

Referenz : Kapitel III. Eingabe und Bearbeitung von Texten

Kommando : **BackSpace**
Abkürzung : **Back**
Parameter : -
Funktion :
Löscht das Zeichen links vom Cursor. Steht der Cursor am Anfang einer Zeile, so wird diese mit der vorigen Zeile zusammengefügt.

Kommando : **Delete**
Abkürzung : **Del**
Parameter : -
Funktion :
Löscht das Zeichen, auf dem sich der Cursor befindet. Steht der Cursor am Ende einer Zeile, so wird diese mit der folgenden Zeile zusammengefügt.

Kommando : **KillLine**
Abkürzung : **Kill**
Parameter : -
Funktion :
Entfernt die aktuelle Zeile aus dem Text. Sie wird im Undo-Buffer gespeichert.

Kommando : **UndoLine**
Abkürzung : **Undo**
Parameter : -
Funktion :
Befindet sich der Cursor in der zuletzt geänderten Zeile, so wird diese in ihren alten Zustand vor der Änderung versetzt.
Befindet sich der Cursor in einer anderen Zeile, so wird die zuletzt geänderte Zeile in ihrem Zustand vor der Änderung vor der aktuellen Zeile eingefügt. Dieses Verhalten dient vor allen dazu, eine durch „KillLine“ gelöschte Zeile wieder richtig in den Text einzufügen.

Kommando : **IndentReturn**
Abkürzung : **IRet**
Parameter : -
Funktion :
Die aktuelle Zeile wird an der Cursorposition in zwei Zeilen aufgespalten. Der Teil der aktuellen Zeile, rechts vom Cursor , wird zur neuen aktuellen Zeile und mit Leerzeichen bzw. Tabs genau wie die davorliegende Zeile (die alte aktuelle) eingerückt.

Kommando : **Return**
Abkürzung : **Ret**
Parameter : -
Funktion :
Die aktuelle Zeile wird an der Cursorposition in zwei Zeilen aufgespalten. Der Teil der aktuellen Zeile, rechts vom Cursor, wird zur neuen aktuellen Zeile.

Kommando : **BigStepUp**
Abkürzung : **BSUp**
Parameter : -
Funktion :
Bewegt den Cursor um etwas weniger als die Anzahl der angezeigten Zeilen nach oben.

Kommando : **BigStepDown**
Abkürzung : **BSDown**
Parameter : -
Funktion :
Bewegt den Cursor um etwas weniger als die Anzahl der angezeigten Zeilen nach unten.

Kommando : **MoveUp**
Abkürzung : **Up**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeilen nach oben.

Kommando : **MoveDown**
Abkürzung : **Down**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeilen nach unten.

Kommando : **ScreenUp**
Abkürzung : **SUp**
Parameter : p_0=VALUE
Funktion :
Verschiebt den Textausschnitt unter dem Cursor um p_0 Zeilen nach unten.

Kommando : **ScreenDown**
Abkürzung : **SDown**
Parameter : p_0=VALUE
Funktion :
Verschiebt den Textausschnitt unter dem Cursor um p_0 Zeilen nach oben.

Kommando : **MoveLeft**
Abkürzung : **Left**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeichen nach links. Geht der Cursor über den Anfang einer Zeile hinaus, so wird er an das Ende der davorliegenden Zeile gesetzt.

Kommando : **MoveRight**
Abkürzung : **Right**
Parameter : p_0=VALUE
Funktion :
Bewegt den Cursor um p_0 Zeichen nach rechts. Geht der Cursor über das Ende einer Zeile hinaus, so wird er an den Anfang der folgenden Zeile gesetzt.

Kommando : **GoStringPos**
Abkürzung : **GoSPos**
Parameter : p_0=VALUE
Return : -
Funktion :
Positioniert den Cursor auf das Zeichen an Position p_0 der Zeichenkette der aktuellen Zeile. Die Position wird von 0 an gezählt.

Kommando : **GoDispPos**
Abkürzung : **GoPos**
Parameter : p_0=VALUE
Return : -
Funktion :
Positioniert den Cursor in der aktuellen Zeile auf Position p_0. p_0 wird in diesem Fall von 1 an gezählt und korrespondiert mit der Anzeige in der Titelleiste des Fensters.

Kommando : **Top**
Abkürzung : **T**
Parameter : -
Funktion :
Setzt den Cursor auf den Anfang der ersten Zeile des Textes.

Kommando : **Bot**
Abkürzung : **B**
Parameter : -
Funktion :
Setzt den Cursor auf den Anfang der letzten Zeile des Textes.

II. Verwalten von Texten

Referenz : Kapitel IV. Text

Kommando : **NewText**
Abkürzung : -
Funktion :
Erzeugt einen neuen Text.

Kommando : **FreeText**
Abkürzung : **FTx**
Parameter : -
Funktion :
Gibt den aktuellen Text frei.

Kommando : **PredText**
Abkürzung : **PTx**
Parameter : -
Funktion :
Wählt den Text vor dem aktuellen Text an oder, falls dieser der erste Text in der Liste war, den letzten.

Kommando : **SuccText**
Abkürzung : **STx**
Parameter : -
Funktion :
Wählt den Text nach dem aktuellen Text an oder, falls dieser der letzte Text in der Liste war, den ersten.

Kommando : **SelTextN**
Abkürzung : **STN**
Parameter : p_0=VALUE
Funktion :
Wählt Text Nummer p_0 aus der Liste vorhandener Texte aus.

Kommando : **New**
Abkürzung : **New**
Parameter : -
Funktion :
Löscht alle Zeilen des aktuellen Textes.

Kommando : **Print**
Abkürzung : **Pr**
Parameter : p_0=STRING
Funktion :
Der aktuelle Text wird in die durch p_0 angegebene Datei ausgegeben. Durch Angabe von p_0="ptr" wird auf den durch die Preferences eingestellten Drucker gedruckt.

III. Dateioperationen

Referenz : Kapitel V. Dateioperationen

Kommando : **Save**
Abkürzung : **SA**
Parameter : p_0=OPT_STRING
Funktion :
Speichern des aktuellen Textes.
Wurde p_0 angegeben, so wird der Text unter diesem Namen abgespeichert und als neuer Name des Textes übernommen. Wurde für p_0 ein Leer-String angebenen, so wird der Name des Textes über einen Dateiauswahlfenster vom Benutzer abgefragt.
Wird p_0 nicht mitangegeben, so wird der Text unter seinem bisherigen Namen abgespeichert.

Kommando : **Open**
Abkürzung : **Op**
Parameter : p_0=OPT_STRING
Funktion :
Überladen des aktuellen Textes. Der aktuelle Text wird „vergessen", und der durch p_0 angegebene Text wird geladen. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name des Textes über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **InsertFile**
Abkürzung : **InF**
Parameter : p_0=OPT_STRING
Funktion :
Einfügen eines Textes in den aktuellen Text. Zu p_0 gilt das bereits oben zu „Save" und „Open" Gesagte.

Kommando : **UpdateText**
Abkürzung : **UtX**
Parameter : p_0=Value
Funktion :
Speichert den Text mit der ID p_0, falls er geändert aber noch nicht gespeichert wurde.

Kommando : **UpdateTexts**
Abkürzung : **UtXs**
Parameter : -
Funktion :
Speichert jeden Texte, der geändert aber noch nicht abgespeichert wurde.

IV. Blockoperationen

Referenz : Kapitel VI. Blockoperationen

Kommando : **Mark**
Abkürzung : **Mk**
Parameter : -
Funktion :
Aktiviert den Markierungsmodus für normale Blöcke. Die Cursorposition bei Ausführung von „Mark" wird als Block-Anfang benutzt. Die aktuelle Cursorposition bei Ausführung einer Blockoperation gilt als Block-Ende.
Erneutes Ausführen von „Mark" bewirkt eine Desaktivierung des Markierungsmodus.

Kommando : **MarkVert**
Abkürzung : **MV**
Parameter : -
Funktion :
Aktiviert den Markierungsmodus für Spaltenblöcke. Erneutes Ausführen von „MarkVert" bewirkt eine Desaktivierung des Markierungsmodus.

Kommando : **Cut**
Abkürzung : **X**
Parameter : -
Funktion :
Schneidet dem markierten Block aus dem Text aus und kopiert ihn in das ClipBoard. Beendet den Blockmarkierungsmodus. Der Text vor dem ausgeschnittenen Bereich wird mit dem Text nach dem ausgeschnittenen Bereich zusammengefügt.

Kommando : **Copy**
Abkürzung : **C**
Parameter : -
Funktion :
Kopiert den markierten Block in das ClipBoard. Beendet den Blockmarkierungsmodus.

Kommando : **Paste**
Abkürzung : **V**
Parameter : -
Funktion :
Fügt den im ClipBoard gespeicherten Text an der momentanen Cursorposition ein. Anschließend wird der Cursor auf das letzte eingefügte Zeichen positioniert.

Abkürzung : **PV**
Parameter : -
Funktion :
Fügt die im ClipBoard gespeicherte Textspalte korrekt an der momentanen Cursorposition ein. Anschliessend wird der Cursor auf das letzte eingefügte Zeichen positioniert.

Kommando : **SaveClip**
Abkürzung : **SC**
Parameter : p_0=OPT_STRING
Funktion :
Speichert Text aus dem ClipBoard in einer Datei mit Namen p_0. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **OpenClip**
Abkürzung : **OC**
Parameter : p_0=OPT_STRING
Funktion :
Lädt Text aus der durch p_0 angebenen Datei und kopiert ihn in das ClipBoard. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **BlockSL**
Abkürzung : **BSl**
Parameter : -
Funktion :
Rückt alle markierten Zeile eines Blocks um eine Tabulatorweite nach links, falls vor jeder Zeile mindestens ein Tab-Zeichen, bzw. entsprechend viele Leerzeichen sind.

Kommando : **BlockSR**
Abkürzung : **BSr**
Parameter : -
Funktion :
Rückt alle markierten Zeile eines Blocks um eine Tabulatorweite nach rechts.

Kommando : **BlockLower**
Abkürzung : **BLo**
Parameter : -
Funktion :
Wandelt alle markierten Zeichen eines Blocks in Kleinbuchstaben um.

Kommando : **BlockHigher**
Abkürzung : **BHi**
Parameter : -
Funktion :
Wandelt alle markierten Zeichen eines Blocks in Großbuchstaben um.

V. Suchen und Ersetzen

Referenz : Kapitel VII. Suchen und Ersetzen

Kommando : **Find**
Abkürzung : **f**
Parameter : p_0=STRING
Funktion :
Sucht ausgehend von der aktuellen Cursorposition die durch p_0 angebene Zeichenkette im Text. Dabei wird solange in eine Richtung gesucht bis ein Vorkommen der Zeichenkette gefunden werden konnte. Ansonsten wird die Suchrichtung umgekehrt und abgebrochen.

Kommando : **Replace**
Abkürzung : **r**
Parameter : p_0=STRING, p_1=STRING
Funktion :
Sucht ausgehend von der aktuellen Cursorposition die durch p_0 angebene Zeichenkette im Text. Dabei wird solange in eine Richtung gesucht bis ein Vorkommen der Zeichenkette gefunden werden konnte. Ansonsten wird die Suchrichtung umgekehrt und abgebrochen.
Konnte p_0 gefunden werden, so wird sie durch die mit p_1 angegebene Zeichenkette ersetzt.

Kommando : **Again**
Abkürzung : **a**
Parameter : -
Funktion :
Wiederholt die letzte Such- bzw. Ersetzungsoperation.

Kommando : **ReplaceAll**
Abkürzung : **ra**
Parameter : p_0=STRING, p_1=STRING
Funktion :
Ersetzt alle Vorkommen von p_0 im Text durch die Zeichenkette p_1.

Kommando : **RemTab**
Abkürzung : **rt**
Parameter : -
Funktion :
Ersetzt alle Tabs im Text durch entsprechend viele Spaces.

Kommando : **GoMatchStruct**
Abkürzung : **GMS**
Parameter : -
Funktion :
Der Cursor muß sich vor Ausführung von „GoMatchStruct“ auf einer beliebigen öffnenden oder schließenden Klammer befinden. „GoMatchStruct“ bewegt den Cursor dann auf die korrespondieren Klammer.

Kommando : **GoParsedError**
Abkürzung : **GPE**
Parameter : -
Funktion :
Der Cursor muß sich auf der Zeile einer von einem Compiler erzeugten Fehlerdatei befinden. in welcher der Dateiname und die Zeilennummer eines Fehlers stehen. „GoParsedError“ analysiert diese Zeile und prüft, ob die Datei, zu der die Fehlerbeschreibung gehört, sich schon im Speicher befindet. Ist dies nicht der Fall, so wird die entsprechende Datei nachgeladen. Daraufhin wird der Cursor auf die Zeile positioniert, in welcher sich der angebene Fehler befindet.

Kommando : **Jump**
Abkürzung : **J**
Parameter : p_0=VALUE
Funktion :
Positioniert den Cursor auf die durch p_0 angegebene Zeile.

VI. Strukturieren von Texten

Referenz: Kapitel VIII. Strukturieren von Texten

Kommando : **Fold**
Abkürzung : **Fo**
Parameter : -
Funktion :
Faltet einen markierten Block zusammen.

Kommando : **ShowFold**
Abkürzung : **SF**
Parameter : -
Funktion :
Befindet sich der Cursor auf einer geschlossenen Falte, so wird diese hierdurch geöffnet.

Kommando : **RemFold**
Abkürzung : **rf**
Parameter : -
Funktion :
Befindet sich der Cursor auf der ersten Zeile einer Falte, so wird die Falte durch „RemFold" geöffnet, und die Faltenmarkierungen werden entfernt.

Kommando : **SetFoldsOff**
Abkürzung : **SF0**
Parameter : -
Funktion :
Sorgt dafür, daß geschlossene, gefaltete Bereiche mit allen Zeilen genau wie geöffnete, gefaltete Bereiche dargestellt werden.

Kommando : **SetFoldsOn**
Abkürzung : **SF1**
Parameter : -
Funktion :
Macht „SetFoldsOff" rückgängig. Danach werden nur noch explizit mit „ShowFold" geöffnete Falten auch als geöffnet dargestellt.

Kommando : **GoPredFold**
Abkürzung : **GPF**
Parameter : -
Funktion :
Bewegt den Cursor auf den nächstliegenden Faltenanfang oder das nächstliegende Faltenende vor der aktuellen Zeile.

Kommando : **GoSuccFold**
Abkürzung : **GSF**
Parameter : -
Funktion :
Bewegt den Cursor auf den nächstliegenden Faltenanfang oder das nächstliegende Faltenende nach der aktuellen Zeile.

Kommando : **ToggleKey**
Abkürzung : **TK**
Parameter : -
Funktion :
Ändert den Auszeichnungsstatus der aktuellen Zeile. Ist diese ausgezeichnet, so wird die Auszeichnung gelöscht und umgekehrt.

Kommando : **GoPredKey**
Abkürzung : **GPK**
Parameter : -
Funktion :
Bewegt den Cursor auf die nächste vor der aktuellen Zeile liegende Zeile, die ausgezeichnet ist.

Kommando : **GoSuccKey**
Abkürzung : **GSK**
Parameter : -
Funktion :
Bewegt den Cursor auf die nächste der aktuellen Zeile folgende Zeile, die ausgezeichnet ist.

Kommando : **SetBookMark**
Abkürzung : **SBM**
Parameter : p_0=VALUE
Funktion :
Belegt das Lesezeichen mit der Nummer p_0 mit einem Verweis auf die aktuelle Zeile, so daß diese einfach wieder angesprungen werden kann.

Kommando : **GoBookMark**
Abkürzung : **GBM**
Parameter : p_0=VALUE
Funktion :
Sucht, ob das Lesezeichen mit der Nummer p_0 einen Verweis auf eine Zeile enthält und positioniert dann den Cursor auf diese.

VII. Ändern von Optionen, die den aktuellen Text betreffen

Referenz : Kapitel IX. Das Edit-Prefs Formular

Kommando : **SetTab**
Abkürzung : -
Parameter : p_0=VALUE
Funktion :
Setzt die Tabulator-Weite auf p_0. p_0 muß im Bereich von 1 bis 20 liegen.

Kommando : **SetOverOn**
Abkürzung : **SO1**
Parameter : -
Funktion :
Aktiviert den „OverWrite"-Modus.

Kommando : **SetOverOff**
Abkürzung : **SO0**
Parameter : -
Funktion :
Desaktiviert den „OverWrite"-Modus.

Kommando : **SetWrap**
Abkürzung : **SWr**
Parameter : p_0=VALUE
Funktion :
Setzt den rechten Rand für den Umbruch im „Word-Wrap"-Modus auf p_0. p_0 muß größer als 9 sein.

Kommando : **SetWrapOn**
Abkürzung : **sw1**
Parameter : -
Funktion :
Aktiviert den „Word-Wrap"-Modus.

Kommando : **SetWrapOff**
Abkürzung : **sw0**
Parameter : -
Funktion :
Desaktiviert den „Word-Wrap"-Modus.

VIII. „Word-Wrap“

Referenz : Kapitel X. „Word-Wrap“

Kommando : **WrapPara**
Abkürzung : **WrP**
Parameter : -
Funktion :
Formatiert den aktuellen Absatz gemäß den Einstellungen für den „Word-Wrap“ um.

Kommando : **WrapText**
Abkürzung : **WrT**
Parameter : -
Funktion :
Formatiert den aktuellen Text gemäß den Einstellungen für den „Word-Wrap“ um.

XI. Makros

Referenz : Kapitel XI. Makros

Kommando : **RecStart**
Abkürzung : **rst**
Parameter : -
Funktion :
Leitet die Aufzeichnung eines Makros ein. Das zuletzt aufgezeichnete Makro geht dadurch verloren.

Kommando : **RecStop**
Abkürzung : **rsp**
Parameter : -
Funktion :
Beendet die Aufzeichnung eines Makros.

Kommando : **RecPlay**
Abkürzung : **RP**
Parameter : -
Funktion :
Spielt ein aufgezeichnetes oder eingeladenes Makro ab.

Kommando : **SaveRecScript**
Abkürzung : **SRS**
Parameter : p_0=OPT_STRING
Funktion :
Speichert ein aufgezeichnetes Makro in einer Datei mit Namen p_0 ab.
Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **PlayRecScript**
Abkürzung : **PRS**
Parameter : p_0=OPT_STRING
Funktion :
Lädt die durch p_0 angegebene Makro-Datei ein, interpretiert diese und führt sie aus. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

X. Fenster

Referenz : Kapitel XII. Fenster

Kommando : **NewWin**
Abkürzung : **NW**
Parameter : -
Funktion :
Eröffnet ein neues Fenster.

Kommando : **FreeWin**
Abkürzung : **FW**
Parameter : -
Funktion :
Schließt das aktuelle Fenster. Falls das aktuelle Fenster das letzte Fenster Edwards war, so wird Edward beendet.

Kommando : **PredWin**
Abkürzung : **PW**
Parameter : -
Funktion :
Wählt das Window vor dem aktuellen Window an oder, falls dieses das erste Window in der Liste war, das letzte.

Kommando : **SuccWin**
Abkürzung : **SW**
Parameter : -
Funktion :
Wählt das Window nach dem aktuellen Window an oder, falls dieses das letzte Window in der Liste war, das erste.

Kommando : **SplitHorWin**
Abkürzung : **SHW**
Parameter : -
Funktion :
Teilt das aktuelle Fenster horizontal in zwei gleichgroße Fenster auf.

Kommando : **SplitVerWin**
Abkürzung : **SVW**
Parameter : -
Funktion :
Teilt das aktuelle Fenster vertikal in zwei gleichgroße Fenster auf.

Kommando : **FullSizeWin**
Abkürzung : **FSW**
Parameter : -
Funktion :
Vergrößert das aktuelle Fenster auf maximale Ausmaße.

XI. Verlassen des Editors

Referenz : Kapitel XIII. Verlassen des Editors

Kommando : **Exit**
Abkürzung : **XIt**
Parameter : -
Funktion :
Edward verlassen. Informationen zur Aufnahme der Arbeit in derselben Umgebung, wie vor dem Aufruf von „Exit“ werden gespeichert.

Kommando : **Quit**
Abkürzung : **Q**
Parameter : -
Funktion :
Edward sofort verlassen.

XII. Formulare

Referenz : Kapitel VII. Suchen und Ersetzen
Kapitel IX. Das Edit-Prefs Formular
Kapitel XVI. Globale Einstellungen

Kommando : **FindForm**
Abkürzung : **FFm**
Parameter : -
Funktion :
Öffnet das Such-Formular.

Kommando : **FindRepForm**
Abkürzung : **FRFm**
Parameter : -
Funktion :
Öffnet das Formular zum Suchen und Ersetzen.

Kommando : **GotoForm**
Abkürzung : **GFm**
Parameter : -
Funktion :
Öffnet das Formular, mit welchem eine spezielle Zeile angesprungen werden kann.

Kommando : **GlobalForm**
Abkürzung : **GFm**
Parameter : -
Funktion :
Öffnet das Formular für die globalen Einstellungen.

Kommando : **PageForm**
Abkürzung : **PFm**
Parameter : -
Funktion :
Öffnet das Formular, in welchem den Ausdruck von Texten betreffende Einstellungen vorgenommen werden können.

Kommando : **EditForm**
Abkürzung : **EdFm**
Parameter : -
Funktion :
Öffnet das Formular, mit dem den aktuellen Text betreffende Einstellungen vorgenommen werden können.

Kommando : **InfoForm**
Abkürzung : **InfFm**
Parameter : -
Funktion :
Öffnet das Formular mit Informationen über Edward.

XIII. Der Kommandozeilen-Interpreter

Referenz : Kapitel XV. Der Kommandozeilen-Interpreter

Kommando : **RepeatCmd**
Abkürzung : **RpCmd**
Parameter : -
Funktion :
Führt die zuletzt eingegebene Kommandozeile erneut aus.

Kommando : **EnterCmd**
Abkürzung : **ECmd**
Parameter : -
Funktion :
Wechselt in den Eingabemodus für die Kommandozeile. Ist dieser Eingabemodus aktiv, so bricht erneutes Ausführen von „EnterCmd" den Eingabemodus ab, ohne daß die eingebene Zeile ausgeführt wird.

Kommando : **ModifyOldCmd**
Abkürzung : **MCmd**
Parameter : -
Funktion :
Wechselt in den Eingabemodus für die Kommandozeile, wobei die zuletzt eingegebene Kommandozeile erneut zur Bearbeitung bereitgestellt wird. Ist dieser Eingabemodus aktiv, so bricht erneutes Ausführen von „ModifyOldCmd" den Eingabemodus ab, ohne daß die eingebene Zeile ausgeführt wird.

Kommando : **Do**
Abkürzung : **Do**
Parameter : p_0=VALUE, p_1=CMDLIST
Funktion :
Führt die durch p_1 angegebene Kommandofolge p_0-mal aus.

XVI. Ändern von globalen Einstellungen

Referenz: Kapitel XVI. Globale Einstellungen
Kapitel XX. Benutzerdefinierte Menüs, Tastaturbelegung und HotKey

Kommando : **Display**
Abkürzung : **Dsp**
Parameter : p_0=VALUE
Funktion :
Durch „Display" wird der Screen gewählt, auf welchem die Fenster von Edward erscheinen sollen. Dabei sind die Werte von p_0 folgendermaßen zugeordnet :

0 = Workbench-Screen
1 = 4 farbiger Screen, Hires
2 = 4 farbiger Screen, Hires-Interlaced

Kommando : **Font**
Abkürzung : **Fnt**
Parameter : p_0=STRING, p_1=VALUE
Funktion :
p_0 muß den Namen eines Zeichensatzes angeben (Bsp. „topaz.font"). p_1 gibt die gewünschte Größe des Zeichensatzes an. Bei der Auswahl der Zeichensätze gelten gewisse Einschränkungen. So sind z.B. keine proportionalen Zeichensätze möglich. Sollte Edward aus irgendeinem Grund einen gewünschten Zeichensatz für ungeeignet halten, so wird der zuvor eingestellte Zeichensatz weiterhin verwendet.

Kommando : **AutoMode**
Abkürzung : **Auto**
Parameter : p_0=VALUE, p_1=OPT_VALUE
Funktion :
p_0 wählt das Verfahren für den Auto-Save, p_1 gibt optional die Zeitspanne in Sekunden zwischen zwei Sicherungen an. p_1 muß einen Wert größer oder gleich 30 haben.

Die Werte von p_0 entsprechen folgenden Modi :
0 = Auto-Save OFF
1 = Auto-Save (automatisch speichern, ohne Bestätigung)
2 = Confirm Auto-Save (automatisch speichern, aber erst nach Bestätigung)

Kommando : **OpenDefs**
Abkürzung : **OD**
Parameter : p_0=OPT_STRING
Funktion :
Die Definition der Menüs, Tastaturbelegung, des HotKeys, etc. wird aus der durch den String p_0 spezifizierten Datei geladen. Wird kein String angegeben, so öffnet sich ein Dateiauswahlfenster, mit dem eine Datei gewählt werden kann.

XV. ARexx - Kommunikation mit Edward

Referenz : Kapitel XXI. ARexx - Kommunikation mit Edward

Kommando : **ExecRexx**
Abkürzung : **RX**
Parameter : p_0=OPT_STRING
Funktion :
Führt die durch p_0 angebene Datei als Rexx-Programm aus. Wird p_0 nicht oder nur ein Leer-String für p_0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **ExecRexxString**
Abkürzung : **RXS**
Parameter : p_0=STRING
Funktion :
Führt den übergebenen String p_0 als kleines Rexx-Programm aus.

1) Abfrage von Statuswerten Edwards :

Kommando : **GetWord**
Abkürzung : **GWd**
Parameter : -
Return : SMALL_STRING
Funktion :
Liefert das Wort, auf welchem sich der Cursor befindet zurück.

Kommando : **GetLineString**
Abkürzung : **LineString**
Parameter : -
Return : STRING
Funktion :
Gibt den Inhalt der aktuellen Zeile zurück.

Kommando : **GetTextPos**
Abkürzung : **TextPos**
Parameter : -
Return : LIST „(LineNum, DispPos, DispLineNum, EdStringPos, Fold, FoldStart, FoldEnd, Key, EndLF, EndEOS)“
Funktion :
Liefert Liste, die die Position im Dokument und Status-Informationen enthält.

Aufbau der Liste :

- Zeilennummer; ergibt sich durch Abzählen aller Zeilen vom Textanfang an.
- Spaltenposition
- Zeilennummer; ergibt sich durch Abzählen nur der sichtbaren Zeilen (d.h.: die eingefalteten Zeilen werden nicht mitgezählt).
- Position im String der aktuellen Zeile, die mit der Spaltenposition korrespondiert. Enthält ein String TAB-Zeichen, so können sich die Spaltenposition und die String-Position unterscheiden, da ein Tab für die Spaltenposition normalerweise wie mehrere Leerzeichen gezählt wird.
- Fold. Flag; gibt an, ob die Zeile eingefaltet ist.
- FoldStart. Flag; gibt an, ob die Zeile der Beginn einer Falte ist.
- FoldEnd. Flag; gibt an, ob die Zeile das Ende einer Falte ist.
- Key. Flag; gibt an, ob die Zeile ausgezeichnet wurde.
- EndLF. Flag; gibt an, ob die Zeile mit einem LineFeed (ASCII 10) endet.
- EndEOS. Flag; gibt an, ob die Zeile mit einem EOS (ASCII 0) endet.

Kommando : **GetTextList**
Abkürzung : **TextList**
Parameter : -
Return : LIST „([TextName ID]*)“
Funktion :
Liefert eine Liste aller geladen Texte und ihrer IDs.

Kommando : **GetWindowList**
Abkürzung : **WindowList**
Parameter : -
Return : LIST „([WindowID TextID Left Top Width Height]*)“
Funktion :
Liefert eine Liste mit IDs aller geöffneten Windows, sowie dazugehörigen Informationen (ID des dargestellten Textes, Position und Ausmaße des Fensters).

Kommando : **GetBlockInfo**
Abkürzung : **BlockInfo**
Parameter : -
Return : LIST „(Block, MarkVert, LineNum, DispPos, StringPos)“
Funktion :
Gibt Informationen über einen möglicherweise markierten Block zurück. Ein markierter Block wird immer durch die zu Beginn markierte Zeile und die aktuelle Zeile definiert.

Die Liste enthält folgende Werte:

- Block; Flag. Gibt an, ob ein Block markiert ist. Ist dieses Flag 0, so gelten alle folgenden Informationen nicht.
- MarkVert; Flag. Gibt an, daß der Benutzer einen Spaltenblock markieren will.
- LineNum. Zeilennummer der Zeile, an der die Markierung beginnt (diese kann über-, aber auch unterhalb der aktuellen Zeile liegen).
- DispPos. Spaltenposition des Markierungsbeginns.
- StringPos. Zu DispPos korrespondierende Position im String der Zeile, an der die Markierung beginnt.

Kommando : **GetTextInfo**
Abkürzung : **TextInfo**
Parameter : -
Return : LIST „(TotalLines, DispLines, FoldedLines, UseFold, DoWrap, Over, LockCnt, Changes.)“
Funktion :
Liefert allgemeine Informationen über den aktuellen Text.

Aufbau der Liste :

- TotalLines; Anzahl aller Zeilen des Textes
- DispLines; Anzahl der momentan darstellbaren Zeilen.
- FoldedLines; Anzahl der eingefalteten Zeilen.
- UseFold; Flag. Gibt an, ob Zeilen innerhalb von Falten dargestellt werden sollen (0) oder nicht(-1).
- DoWrap; Flag. Gibt an, ob der WordWrap-Modus eingeschaltet ist.
- Over; Flag. Gibt an, ob der OverWrite-Modus eingeschaltet ist.
- LockCnt; Zähler, wie oft der Text gelocked wurde. Ist er ungleich 0, so kann der Text nicht modifiziert werden. Der LockCnt wird hochgezählt, wenn der Text gedruckt werden soll, oder TextReads auf den Text ausgeführt werden.

Kommando : **GetScreenInfo**
Abkürzung : **ScreenInfo**
Parameter : -
Return : LIST „(Width Height Depth ScreenAdr)“
Funktion :
„ScreenInfo“ gibt folgende Werte in einer Liste zurück :
Breite, Höhe, Tiefe und Adresse des Screens, auf welchem Edward seine Fenster darstellt.

2) Laden von Texten aus Edward heraus

Mit den folgenden Befehlen ist ein Zugriff auf im Speicher befindliche Texte Edwards auf einfache und schnelle Weise möglich.
Zu beachten ist hier vor allem, daß ein von einer externen Applikation mit InitTextRead geöffneter Text bis zum Abschluss der TextRead- Operationen durch EndTextRead nicht verändert werden kann; der Text ist gesperrt („gelocked“).
Solange nicht alle InitTextReads durch entsprechende EndTextReads abgeschlossen wurden, kann zudem Edward nach einem Beenden nicht aus dem Speicher entfernt werden.
Es ist selbstverständlich möglich, mehrere InitTextReads auf denselben Text auszuführen.

Kommando : **InitTextRead**
Abkürzung : **ITR**
Parameter : p_0=STRING
Return : Handle=VALUE
Funktion :
Versucht Text mit Namen p_0 zu finden. Ist dieser vorhanden, so wird er gelocked und ein Handle für weitere Operationen erzeugt. Die erste Zeile des Textes wird zur aktuellen Zeile für kommende TextRead-Operationen. Das Handle wird zurückgegeben.

Kommando : **TextRead**
Abkürzung : **TR**
Parameter : Handle=VALUE, Buffer=VALUE, BuffSize=VALUE
Return : RealRead=VALUE
Funktion :
Kopiert von der für diesen TextRead aktuellen Zeile den Inhalt dieser und der folgenden Zeilen in den Buffer. Dabei werden Zeilen immer komplett kopiert. Sollte eine Zeile nicht mehr ganz in den Buffer passen (daher muß BuffSize angegeben werden), so wird die Kopieraktion beendet.
Daraufhin wird die Anzahl der kopierten Bytes zurückgegeben (RealRead). Das Lesen ist beendet, wenn RealRead gleich 0 ist.

Kommando : **EndTextRead**
Abkürzung : **ETR**
Parameter : Handle=VALUE
Return : -
Funktion :
Der durch ein „InitTextRead" gelockte Text wird wieder unlocked. Das Handle wird freigegeben.

XVI. UnDo/ReDo - Zurücknehmen von Text-Änderungen

Referenz: Kapitel XXII. Zurücknehmen von Text-Änderungen

Kommando : **UnDo**
Abkürzung : **UnDo**
Parameter : -
Funktion :
Nimmt die letzte Änderung am Text zurück.

Kommando : **ReDo**
Abkürzung : **ReDo**
Parameter : -
Funktion :
Nimmt die Rücknahme der Rückname der letzten Textänderung zurück.
(Gut, das ist nicht gerade leichverständlich, aber Sie können ja noch einmal in das Referenzkapitel blättern. Noch besser ist, Sie probieren es einfach aus.)

XVII. Verschiedenes

Referenz : Kapitel XIV. Iconify & HotKey

Kommando : **ExecCli**
Abkürzung : **Exe**
Parameter : p_0=STRING
Funktion :
Führt das Programm mit Namen p_0 aus.

Kommando : **InsertChars**
Abkürzung : **IC**
Parameter : p_0=STRING
Funktion :
Fügt die durch p_0 spezifizierten Zeichen an der aktuellen Cursorposition ein. Dieses Kommando wurde für Makros und die ARexx-Kommunikation eingeführt.

Kommando : **SetMode1, SetMode2, SetMode3**
Abkürzung : **SM1, SM2, SM3**
Parameter : -
Funktion :
Sorgt dafür, daß die nächste Tastatureingabe gemäß Modus 1 interpretiert wird.

Kommando : **Iconify**
Abkürzung : **Ify**
Parameter : -
Funktion :
Schließt alle Fenster Edwards. Ein neues Fenster, das nur aus einer Titelzeile besteht erscheint anschließend. Wird dieses Fenster aktiviert und die rechte Maustaste gedrückt, so werden alle Fenster Edwards wieder an ihren ursprünglichen Positionen geöffnet. Edward kann ebenfalls durch den HotKey oder das Senden einer Nachricht über ARexx „geweckt“ werden.

Anhang B) Fehlermeldungen.

1) Informationen

2) Allgemeine Fehlermeldungen

3) Fehlermeldungen mit Abfragen

4) Meldungen mit Abfragen

5) Fehler bei Betriebssystemoperationen

6) Eingabefehler

7) Fehler in den Definitionsdateien

8) Fehler beim Analysieren von Kommandos

9) Fehlermeldungen beim Auswerten von Kommandos

Beim Arbiten mit Edward werden Sie bei wichtigen Problemen, die sich ergeben können, augenblicklich informiert und erhalten in den meisten Fällen die Möglichkeit, auf diese Informationen zu reagieren. In der Regel handelt es sich hier um Sicherheitsabfragen, um zu verhindern, daß Sie z.B. aus Versehen einen noch nicht abgespeicherten Text löschen.

1) Informationen

Meldung : **„Markierung gesetzt“**

Ursache : Sie haben die aktuelle Zeile mit einem Lesezeichen belegt, um sie demnächst leicht wieder anspringen zu können.

Referenz : Kapitel VIII. 4) Lesezeichen

2) Allgemeine Fehlermeldungen

Meldung : **„Text ist geschützt“**
Ursache : Sie wollten den Text verändern, dieser ist aber gesperrt. Ein Text wird gesperrt, wenn er gedruckt werden soll, oder der Befehl „InitTextRead“ auf ihn ausgeführt wurde.

Abhilfe : Warten, bis der Text gedruckt wurde.
Vielleicht wird auch gerade ein ARexx-Makro auf diesen Text ausgeführt, dann heißt es ebenfalls abwarten.

Referenz : Kapitel IV. 5) Drucken eines Textes
Anhang A) XV.) ARexx - Kommunikation mit Edward

Meldung : **„Kein Pictogramm vorhanden“**

Ursache : Wenn Sie im Formular für die globalen Einstellungen "Configuration Icon" gewählt haben, versucht Edward zu jedem Text, den Sie erzeugen, eine ".info"-Datei zu schreiben. Diese Datei enthält unter anderem Informationen, damit die Workbench ein Pictogramm zu Ihrem Text darstellen kann. Existierte bisher keine ".info"-Datei, so erzeugt Edward eine Standard-Datei. Dazu benötigt er im Verzeichnis "Edward:" ie Datei "EdwardDefault.info". Diese haben Sie anscheinend gelöscht.

Abhilfe : Kopieren Sie von der mitgelieferten Diskette die Datei "EdwardDefault.info" wieder in Ihr "Edward:"-Verzeichnis, sollte Ihnen das zu kompliziert erscheinen müssen Sie Edward komplett neu installieren (Achtung: Dabei können möglicherweise verschiedene von Ihnen vorgenommene Voreinstellungen für Edward verloren gehen).

Referenz : Kapitel XVI. "Globale Einstellungen"

Meldung : **„Abbruch! Konnte Backup nicht erzeugen!“**

Ursache : Sie wollten einen Text speichern. Beim Anlegen der Backup-Datei Ihres Textes trat ein Fehler auf.
Möglicherweise haben Sie einen ungültigenPfad für die Backup-Datei angegeben.
Vielleicht ist auch nur das Laufwerk oder die Platte voll, auf welcher Sie die Backup-Datei anlegen lassen wollten.

Abhilfe : Überprüfen Sie den Pfad-Namen, den Sie für die Backup-Datei im "Global Prefs"-Formular angegeben haben.
Versichern Sie sich, daß auf dem durch den Pfad angegebenen Medium genügend Speicherplatz vorhanden ist.

Referenz : Kapitel XVI. "Globale Einstellungen"

Meldung : **„Markierung nicht gesetzt!“**

Ursache : Sie haben versucht ein Lesezeichen anzuspringen, das noch nicht definiert wurde.

Abhilfe : Vielleicht haben Sie nur die falsche Taste gedrückt, dann sollten Sie es einfach nochmal versuchen.

Referenz : Kapitel VIII. "Strukturieren von Texten - Lesezeichen" Abschnitt 4)

Meldung : **„Kein Block definiert!“**

Ursache : Sie haben versucht eine Block-Operation auszuführen (Cut, Copy, Paste, etc.) , zuvor jedoch keinen Block definiert.

Abhilfe : Einfach den gewünschten Text-Bereich markieren und nochmal versuchen.

Referenz : Kapitel VI. "Blockoperationen"

Meldung : **„Abbruch! Funktion würde Einfaltung zerstören!“**

Ursache : Im allgemeinen wird diese Meldung ausgegeben, wenn Sie einen Block ausschneiden wollten, der nur teilweise eine Text-Falte überdeckt. Bei einer solchen Operation würde nur der Anfang bzw. das Ende der Textfalte ausgeschnitten, was die Falte zerstören würde. Deshalb unterbindet Edward solche Aktionen.

Abhilfe : Sie können eine Textfalte nur komplett ausschneiden. Sollten Sie tatsächlich nur einen Teil der Textfalte ausschneiden wollen, so müssen Sie die Faltung erst rückgängig machen, bevor Sie diesen Teil ausschneiden können.

Referenz : Kapitel VI. "Blockoperationen"

Meldung : **„Zeichenkette zu lang!“**

Ursache : Aus internen Gründen kann Edward maximal 512 Zeichen in einer Zeile verwalten. Beim Einfügen von Zeichen durch eine "Paste"-Operation könnte diese Grenze möglicherweise überschritten werden, was fatale Fehler hervorrufen würde. Daher stoppt Edward hier das Einfügen und gibt obige Meldung aus.

Abhilfe : Die "Paste"-Operation in einer leeren Zeile ausführen.

Referenz : Kapitel VI. "Blockoperationen"

Meldung : **„Suchmuster nicht gefunden!“**

Ursache : Beim Suchen oder Ersetzen konnte das von Ihnen angegebene Suchmuster nirgendwo im Text entdeckt werden.

Abhilfe : Suchvorgang wiederholen. Edward durchsucht den Text immer von der aktuellen Position aus entweder zum Anfang oder zum Ende hin nach dem Suchmuster. Konnte dieses nicht gefunden werden, so kehrt er die Suchrichtung um. Daher kann unter Umständen nach dem Fehlschlagen einer Suchaktion durch eine Wiederholung des Suchvorgangs doch noch etwas gefunden werden. Schlägt dieser Suchvorgang ebenfalls fehl. Dann ist beim besten Willen nichts zu finden.
Vielleicht verbirgt sich eine Zeile, deren Inhalt dem Suchmuster entsprechen würde, in einer Textfalte, dann konnte Edward hier ebenfalls nichts finden. Wählen Sie "Show All Folds" aus dem "Folds"-Menü und wiederholen Sie den Suchvorgang.

Referenz : Kapitel VII. "Suchen und Ersetzen"

Meldung : **„Cursor steht auf keinem Strukturelement“**

Ursache : Beim Aufruf von "Match Bracket" aus dem "Search"-Menü stand der Cursor auf keiner Klammer.

Abhilfe : Positionieren Sie den Cursor auf eine Klammer. Wählen Sie dann erneut "Match Bracket".

Referenz : Kapitel VII. "Suchen und Ersetzen"

Meldung : **„Strukturen stimmen nicht überein!“**

Ursache : Nach dem Aufruf von "Match Bracket" aus dem "Search"-Menü konnte Edward keine korrespondierende öffnende bzw. schließende Klammer finden.

Abhilfe : Überprüfen Sie Ihren Text genau. Sie haben sicher irgendwo eine Klammer vergessen.

Referenz : Kapitel VII. "Suchen und Ersetzen"

Meldung : **„Alles ist rückgängig gemacht worden!“**

Ursache : Sie haben "Undo" aus dem "Extras"-Menü so oft ausgewählt, daß es nichts mehr rückgängig zu machen gibt.

Abhilfe : Wählen Sie doch mal eine andere Funktion aus und bearbeiten Sie Ihren Text wieder.
Na ja, ´mal ganz ernsthaft. Die obige Meldung ist eigentlich eher als Hinweis denn als Fehler zu werten. Nehmen Sie sie einfach zur Kenntnis.

Referenz : Kapitel XXII. "Zurücknehmen von Text-Änderungen"

Meldung : **„Alles ist wiederhergestellt worden!“**

Ursache : Sie haben alle Rücknahmen von Text-Änderungen wieder zurückgenommen, das heißt, Ihr Text sieht wieder genauso aus, wie vor dem ersten Aufruf von "Undo".

Abhilfe : Siehe oben.

Referenz : Kapitel XXII. "Zurücknehmen von Text-Änderungen"

Meldung : **„Menü ist zu groß für den Screen! Nutze Default Einstellungen!“**

Ursache : Sie haben alle Rücknahmen von Text-Änderungen wieder zurückgenommen, das heißt, Ihr Text sieht wieder genauso aus, wie vor dem ersten Aufruf von "Undo".

Abhilfe : Siehe oben.

3) Fehlermeldungen mit Abfragen

Meldung : **„[1][Vorsicht: Kein Speicher mehr!|Texte entfernen|oder Programm verlassen.][OK]“**

Ursache : Edward hat leider nicht genügend Speicher bekommen.

Abhilfe : Sicherheitshalber sollten Sie alle Texte erst einmal abspeichern. Wenn Sie dann einen oder mehrere der momentan nicht mehr benötigten Texte löschen, dann haben Sie für 's erste genügend Speicher zum Weiterarbeiten. Sollte diese Meldung öfter auftreten, dann ist der Kauf einer Speichererweiterung mehr als nur ratsam.

4) Meldungen mit Abfragen

Meldung : **„Geänderten Text verwerfen ?**
[Ja] [Nein]“

Ursache : Sie wollten eine Operation vornehmen, deren Ausführung dazu führen würde, daß der momentan aktuelle Text verloren geht (z.B.: „Open“ aus dem „Projekt“-Menü oder „Neu“ bzw. „Frei“ aus dem „Text“-Menü). Falls dieser Text von Ihnen verändert, aber nicht abgespeichert wurde, so erfolgt diese Meldung. Wenn Sie „Nein“ wählen, so wird die Operation abgebrochen und Sie können den Text danach abspeichern, bei „Ja“ geht dieser Text unwiederbringlich verloren.

Abhilfe : Vor einer der oben erwähnten Operationen den Text abspeichern. Dies ist allerdings nicht zwingend notwendig. Die obige Meldung dient ja gerade dazu, Schlimmeres zu verhindern, wenn Sie das Abspeichern einmal vergessen sollten.

Referenz : Kapitel IV. „Text“, Kapitel V. „Dateioperationen“

Meldung : **„Geänderten Text verwerfen ?**
[Ja] [Nein]“

Ursache : Sie wollten Edward beenden, haben aber noch nicht alle geänderten Texte abgespeichert.

Abhilfe : Wählen Sie „Ja“, dann wird Edward beendet und alle von Ihnen bearbeiteten Texte werden nicht abgespeichert und gehen verloren. Da dies normalerweise nicht gewünscht ist sollten Sie „Nein“ wählen und dann die entsprechenden Texte abspeichern.

Referenz : Kapitel V. „Dateioperationen“

Meldung : **„Altes Pictogramm ersetzen ?**
[Ja] [Nein]“

Ursache : Sie haben eine Datei bearbeitet, zu der ein Icon existiert, das nicht von Edward erzeugt wurde. Dies ist z.B. möglich, wenn Sie eine Script-Datei eines anderen Programmes bearbeiten. Da viele Programme in der „.info“-Datei die das Icon enthält wichtige Zusatzinformationen speichern (diese können Sie sich durch einfaches Anklicken des Icons und Wahl des Menüpunktes „Info“ aus einem Menü der Workbench anzeigen lassen) sollten Sie bei dieser Meldung vorsichtshalber „Nein“ wählen. Nur wenn Sie ganz sicher sind, daß die „.info“-Datei keine wichtigen Informationen enthält wählen Sie „Ja“, dann überschreibt Edward diese Datei mit seiner eigenen „.info“-Datei, die Informationen über das gerade bearbeitete Dokument enthält.

Abhilfe : Vergewissern Sie sich vor der Bearbeitung einer Datei, ob eine „.info“-Datei zu Ihr existiert und ob diese von Edward oder einem fremden Programm erzeugt wurde. Ist letzteres der Fall, so sollten Sie mit dem „Info“-Menü der Workbench erkunden, ob diese Datei irgendwelche wichtigen Zusatzinformationen enthält.

Referenz : Kapitel V. „Dateioperationen“,
Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Datei speichern : 'Name'**
[Ja|Nein]“

Ursache : Sie haben im „Global Prefs“-Formular die Option „Confirm Autosave“ eingestellt. Dies bedeutet, daß Edward nach dem von Ihnen eingestellten Zeitintervall alle Texte überprüft und dann versucht, die geänderten Texte abzuspeichern. Vorher holt er sich jedoch von Ihnen über obige Meldung die Erlaubnis.

Abhilfe : Wenn Sie diese Abfrage stört, dann wählen Sie statt „Confirm Auto-Save“ die Option „Auto-Save“, dann werden geänderte Texte automatisch ohne Rückfrage gespeichert.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das 'Global Prefs'-Formular“

Meldung : **„Datei existiert bereits**
Überschreiben?
[Ja] [Nein]“

Ursache : Vermutlich haben Sie nach Wahl des Menüpunktes „Speichern als...“ aus dem „Projekt“-Menü im Datei-Auswahlfenster den Namen einer Datei gewählt, die schon existiert. Diese würde von momentan aktuellen Text überschrieben, wenn Sie „Ja“ wählen und ginge auf ewig verloren. Um dies zu verhindern erfolgt die Rückfrage Edwards. Wenn Sie sicher sind, daß sie die schon existieren Datei überschreiben wollen, dann wählen Sie „Ja“. Normalerweise ist es wohl sinnvoller „Nein“ zu wählen und sich die entsprechende Datei erst einmal genauer anzuschauen.

Referenz : Kapitel V. „Dateioperationen“

5) Fehler bei Betriebssystemoperationen

Meldung : **„Kein Speicher mehr!“**

Ursache : Sie leiden unter Speicherverknappung. Edward kann in diesem Fall die von Ihnen gewünschte Operation nicht korrekt ausführen.

Abhilfe : Kurzfristig ist es sinnvoll für mehr Speicher zu sorgen, indem sie nicht mehr benötigte Programme die momentan ebenfalls laufen beenden. Weiterhin können Sie Texte, die Sie nicht weiter bearbeiten möchten, mit „Frei“ aus dem „Text“-Menü wieder freigeben und hierdurch ebenfalls etwas Speicher gewinnen.
Langfristig empfiehlt sich wohl die Anschaffung einer RAM-Erweiterung.

Meldung : **„Kann Datei nicht öffnen“**

Ursache : Edward versuchte eine von Ihnen angegebene Datei zum Laden oder Speichern zu öffnen. Dieser Versuch schlug fehl. Eine mögliche Ursache wäre, daß diese Datei momentan von einem anderen Programm bearbeitet wird. Dies ist zum Beispiel möglich, wenn gerade ein Compiler diese Datei übersetzt.

Abhilfe : In oben erwähntem Fall empfiehlt es sich, zu warten, bis der Compiler seine Arbeit beendet hat und dann erneut zu versuchen, diese abzuspeichern. Ansonsten sollten Sie versuchen die Datei unter einem anderen Namen, möglicherweise auch auf einem anderen Laufwerk abzuspeichern.

Meldung : **„Kann nicht speichern“**

Ursache : Beim Schreiben einer Datei schlug irgendetwas fehl.

Abhilfe : Überprüfen Sie, daß das Laufwerk auf welchem Sie die Datei speichern wollten genügen Speicherplatz bietet und kein Schreibschutz aktiviert ist. Sollte dies nicht der Fall sein, so wechseln Sie das Laufwerk, oder deaktivieren den Schreibschutz.

Meldung : **„Kann Info-Datei nicht speichern“**

Ursache : Das Betriebssystem konnte die „.info“-Datei zu Ihrem Text nicht erzeugen.

Abhilfe : s. oben.

Meldung : **„Fehler beim Lesen“**

Ursache : Während des Lesens einer Datei trat ein Fehler auf. Möglicherweise sind Daten auf dem entsprechenden Laufwerk defekt.

Abhilfe : Versuchen Sie mit einem Programm wie DiskDoctor zu retten, was noch zu retten ist. Viel Glück!
Wenn Sie wie empfohlen fleißig Backups Ihrer Arbeiten gemacht haben, dann dürften Sie in dieser Situation nicht viel verloren haben.

Meldung : **„Konnte Datei nicht speichern“**

Ursache : Beim speichern einer Datei schlug irgendetwas fehl.

Abhilfe : s. „Kann nicht speichern“

Meldung : **„Konnte Formular nicht öffnen“**

Ursache : Edward versuchte ein von Ihnen gewünschtes Formular zu öffnen. Dies schien dem Betriebssystem nicht möglich zu sein. Normalerweise ist ein Mangel an freiem Speicher (insbesondere CHIP-Memory) an diesem Verhalten schuld.

Abhilfe : Für freien Speicher sorgen (s. „Keinen Speicher mehr“).

6) Eingabefehler

Meldung : **„Wrap-Position darf nicht kleiner Als 10 sein!“**

Ursache : Sie haben wahrscheinlich in der Kommandozeile eine Wrap-Position kleiner als 10 Zeichen setzen wollen. Da dies nicht sehr sinnvoll ist hat Edward dies unterbunden.

Abhilfe : Eine Position größer oder gleich zehn Zeichen angeben.

Referenz : Kapitel X. „Word-Wrap“

Meldung : **„Tabulatorwert nur zwischen 1 bis 20“**

Ursache : Sie haben versucht einen Wert für die Größe des Tabulatorabstandes anzugeben, der als nicht sinnvoll erachtet wird.

Abhilfe : Sinnvollen Wert zwischen Eins und Zwanzig angeben.

Referenz : Kapitel III. „Eingabe und Bearbeitung von Texten“ Abschnitt 4) RETURN, ENTER und TAB
Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Auflösungen nur im Bereich von 0 bis 2“**

Ursache : Sie versuchten einen ungültigen Wert für den Display-Typ anzugeben.

Abhilfe : Wert zwischen Null und Zwei angeben.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Zeiten kleiner 30 Sek. machen keinen Sinn!“**

Ursache : Die künstliche Intelligenz Edwards hält Intervalle unter dreissig Sekunden für die „Auto-Save“-Funktion für nicht besonders sinnvoll. Wahrscheinlich haben Sie einen Wert unter dreissig Sekunden nur aus Versehen angegeben.

Abhilfe : Werte größer dreissig angeben.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

Meldung : **„Automode-Bereich nur von 0 bis 2!“**

Ursache : Sie haben einen ungültigen Wert für den „Auto-Save“-Modus angegeben.

Abhilfe : Wert zwischen Null und Zwei angeben.

Referenz : Kapitel XVI. „Globale Einstellungen“ Abschnitt 1) „Das ‘Global Prefs’-Formular“

6) Fehler in den Definitionsdateien

Als Referenz zu diesem Abschnitt empfiehlt es sich, das Kapitel XX. „Benutzerdefinierte Menüs, Tastaturbelegung und HotKey“ zu lesen.

Meldung : **„Präprocessor-Befehl zu komplex“**

Ursache : In einer Definitons-Datei, welche die Menüs und die Tastaturbelegung Edwards enthält trat eine übermäßig komplexe Verschachtelung von „#if“-Statements auf. Da Edward hier nur eine Tiefe von maximal 16 beherrscht, hat er hier das Handtuch geworfen.

Abhilfe : Formulieren Sie ihre Definitons-Dateien etwas weniger komplex. Es erscheint schon fast unglaublich, daß Sie selbst diese Schachtelungstiefe verstehen können.

Meldung : **„#endif ohne #if“**

Ursache : Sie haben ein #endif in ihrer Datei, zu dem kein #if gehört.

Abhilfe : Überprüfen Sie die Datei noch einmal genau.

Meldung : **„Präprocessor-Befehl erwartet“**

Ursache : Sie haben in der Datei ein „#“-Zeichen an einer Stelle verwendet, an welcher der Präprozessor dieser bearbeitet. Dieser erwartet nach dem „#“-Zeichen ein Statement („if“, „endif“ oder „else“).

Abhilfe : Überprüfen Sie die Datei. Entfernen Sie das Zeichen, oder ergänzen Sie das entsprechende Statement.

Meldung : **„TITLE erwartet“**

Ursache : Sie haben in der Datei versucht ein Menü zu definieren, daß jedoch keinen einzigen Titel enthält.

Abhilfe : Die entsprechende Stelle der Datei untersuchen. Vielleicht haben Sie den „TITLE“ Befehl ja einfach nur vergessen. Sollten Sie zu den Puristen gehören, die vollständig auf Menüs verzichten können und alles mit der Tastatur machen, dann entfernen Sie auch noch den „MENU“-Befehl aus der Datei.

Meldung : **„Zu viele Menüeinträge“**

Ursache : Sie haben versucht, ein Menü mit mehr als zwanzig Einträgen zu erzeugen. Dies ist leider nicht möglich.

Abhilfe : Die Einträge auf mehrere Menüs verteilen.

Meldung : **„Zu viele Menütitel“**

Ursache : Sie haben versucht, mehr als 16 Menüs zu erzeugen. Dies ist leider nicht möglich.

Abhilfe : Beschränken Sie sich etwas in Ihren Menüs.

7) Fehler beim Analysieren von Kommandos

Fehlermeldungen in Definitions-Dateien oder in Kommandozeilen folgt immer eine Angaben der Position, an welcher der entsprechende Fehler auftrat. Die Fehlermeldungen haben daher folgenden Form:

1. **Fehler**
   In Zeile : Zeile x Column: Spalte

oder

2. **Fehler**
   In Spalte: Spalte

Meldung : **„IDENTIFIER erwartet“**

Ursache : Es wurde ein IDENTIFIER erwartet, aber nicht gefunden.
Ein IDENTIFIER beginnt mit einem Buchstaben, gefolgt von einer beliebigen Anzahl alphanumerischer Zeichen (also Buchstaben und Ziffern) gemischt mit dem Unterstrich „_“.

Meldung : **„STRING erwartet“**

Ursache : Es wurde ein STRING erwartet. Ein String beginnt mit einem Hochkomma („) gefolgt von einer beliebigen Anzahl Zeichen und wird mit einem Hochkomma abgeschlossen.

Meldung : **„NUMBER erwartet“**

Ursache : An dieser Stelle wurde eine Zahl erwartet.

Meldung : **„(erwartet“**

Ursache : Es wurde eine öffnende, runde Klammer erwartet.

Meldung : **„) erwartet“**

Ursache : Eine schliessende, runde Klammer wurde erwartet.

Meldung : **„; erwartet“**

Ursache : Es wurde eine Semikolon erwartet.

Meldung : **„Kein Befehl“**

Ursache : Es wurde ein Kommando aus Edwards Befehlsvorrat erwartet.

Abhilfe : Überprüfen Sie die Stelle. Wahrscheinlich haben Sie den Befehl nur falsch ausgeschrieben.

Meldung : **„LIST oder IDENTIFIER erwartet“**

Ursache : Es wurde eine LIST oder ein IDENTIFIER erwartet, aber nicht gefunden. Eine LIST wird von runden Klammern umschlossen. Zwischen diesen können sich IDENTIFIER, STRINGs, NUMBERs, COMMANDs oder LISTs auch gemischt befinden.

Meldung : **„LOBJECT erwartet“**

Ursache : Es wurde ein LOBJECT erwartet. Ein LOBJECT kann ein STRING, eine NUMBER, ein COMMAND oder eine LIST sein.

8) Fehlermeldungen beim Auswerten von Kommandos

Meldung : **„LVAL erwartet“**

Ursache : Ein LVAL, das bedeutet eine Zahl wurde von einem Kommando als Argument erwartet.

Meldung : **„Zu viele Argumente“**

Ursache : Ein Kommando wurde mit zuvielen Argumenten versorgt.

Meldung : **„Falsches Argument“**

Ursache : Ein Kommando wurde mit falschen Argumenten versorgt.

Sekundärliteratur

- „Das kleine Arschloch“ von Walter Moers, Eichborn Verlag
  Eine Sammlung abgeschmackter Comic-Strips, in denen kein Tabu unberührt bleibt. Hardcore-Humor vom Feinsten. Erklärt außerdem die merkwürdige Vorliebe des Autors dieses Handbuchs für das Wort „Mahlzeit“.

- „Das kleine Arschloch kehrt zurück!“ von Walter Moers, Eichborn Verlag
  Genauso abgeschmackt wie das vorige Buch. Sehr empfehlenswert. „Peppi, peppi, peppi“.

- Das Telefonbuch von Los Angeles.
  Die Telefonnummern der Stars. Rufen Sie z.B. einmal bei Karl Malden an, dem Star aus die „Strassen von San Francisco“. Auch sonst viele interessante Informationen. Sehr entspannende Lektüre.

- „Schlecht ist uns nicht schlecht genug“, von Thomas E. Wieger und Jörg W. Schmidt.
  In mühevoller Kleinarbeit recherchierten die Autoren nach GAGs, den größen anzunehmenden Geschmacklosigkeiten. Alles über die schlechtesten Trash-Fernsehserien der Welt, die ekligsten Nahrungsmittel, die grauenhaftesten Kleidungsstücke, die abstossendsten Schauspieler u.v.a.m..

Epilog

Sehr geehrte Leserin, verehrter Leser,
wer weiß, ob Sie jemals am Ende dieses Handbuches angelangen. Hätte ich zwar eigentlich lieber einen Roman geschrieben, als dieses Handbuch, so hoffe ich doch, daß meine Ausführungen Ihnen die mächtigen Fähigkeiten und Features von Edward nahegebracht haben. Ich bedanke mich für Ihre Aufmerksamkeit und wünsche Ihnen noch viel Erfolg und eine immens gesteigerte Produktivität durch den Einsatz Edwards.
Mahlzeit!

Index

A

Absatz
  neu umbrechen 36
ARexx 61, 65
Auszeichnen 32
Auto-Save 50

B

BACKSPACE 12
Block
  Drucken 23
  Einrücken 23
  Groß- bzw. Kleinbuchstaben 24
  Laden 23
  markieren 20
  Speichern 23
  Umwandeln 24
Blockoperationen 20

C

CLI 10
Copy 20
Cursor-Tasten 13
Cut 20

D

Default-Einstellungen 49
Definition 55
DELETE 12

E

EBNF-Darstellung 53
Edit-Prefs Formular 34
Editor
  Arbeit Unterbrechen 44
  Verlassen 44
EdwardDefault.info 93
EMPTY 56
ENTER 12
Error-Datei 28

F

Falten
auszeichnen ... 33
Entferne ... 32
Entferne alle ... 32
erzeugen ... 31
gefalteten Zeilen anspringen ... 33
rückgängig ... 32
schachteln ... 32
Farben ... 48
Fehlermeldungen
#endif ohne #if ... 104
(erwartet ... 105
) erwartet ... 105
; erwartet ... 106
Alles ist rückgängig gemacht ... 96
Alles ist wiederhergestellt ... 96
Altes Pictogramm ersetzen ... 98
Auflösungen nur im Bereich von 0 bis 2 ... 102
Automode-Bereich nur von 0 bis 2 ... 103
Cursor auf keinem Strukturelement ... 95
Datei existiert bereits ... 99
Falsches Argument ... 106
Fehler beim Lesen ... 101
Geänderten Text verwerfen ... 97
IDENTIFIER erwartet ... 105
Kann Datei nicht öffnen ... 100
Kann Info-Datei nicht speichern ... 101
Kann nicht speichern ... 100
Kein Befehl ... 106
Kein Block definiert ... 94
Kein Pictogramm vorhanden ... 93
Kein Speicher mehr ... 97, 100
Konnte Backup nicht erzeugen ... 93
Konnte Formular nicht öffnen ... 101
LIST oder IDENTIFIER erwartet ... 106
LOBJECT erwartet ... 106
LVAL erwartet ... 106
Markierung gesetzt ... 92
Markierung nicht gesetzt ... 94
Menü ist zu groß ... 96
NUMBER erwartet ... 105
Präprocessor-Befehl erwartet ... 104
Präprocessor-Befehl zu komplex ... 103
STRING erwartet ... 105
Strukturen stimmen nicht überein ... 96
Suchmuster nicht gefunden ... 95
Tabulatorwert nur zwischen 1 bis 20 ... 102
Text ist geschützt ... 93
TITLE erwartet ... 104

Wrap-Position fehlerhaft 102
würde Einfaltung zerstören 94
Zeichenkette zu lang 95
Zeiten kleiner 30 Sek. machen keinen Sinn 103
Zu viele Argumente 106
Zu viele Menüeinträge 104
Zu viele Menütitel 104
Fenster
Erzeugen 40
horizontal teilen 43
mehreren Ansichten 42
Schließen 42
vertikal teilen 43
volle Größe 43
Wahl 41
Find"-Formular 25
Fließtext 36
Fußzeile 51

G

Globale Einstellungen 48
Grenzen 63

H

HotKey 46

I

Iconify 46
Installation 9
interpretieren 28
ITEM 56

K

KeyDef 58
Keys 58
Klammer-Check 27
Kommandos 47, 64
Again 74
AutoMode 85
BackSpace 65
BigStepDown 66
BigStepUp 66
BlockHigher 73
BlockLower 72
BlockSL 72
BlockSR 72
Bot 68
Copy 71
Cut 71

Delete 65
Display 85
Do 85
EditForm 84
EndTextRead 90
EnterCmd 84
ExecCli 91
ExecRexx 86
ExecRexxString 86
Exit 82
Find 74
FindForm 83
FindRepForm 83
Fold 76
Font 85
FreeText 68
FreeWin 81
FullSizeWin 82
GetBlockInfo 88
GetLineString 87
GetScreenInfo 89
GetTextInfo 88
GetTextList 88
GetTextPos 87
GetWindowList 88
GetWord 87
GlobalForm 83
GoBookMark 77
GoDispPos 67
GoMatchStruct 75
GoParsedError 75
GoPredFold 76
GoPredKey 77
GoStringPos 67
GoSuccFold 77
GoSuccKey 77
GotoForm 83
Iconify 91
InfoForm 84
InitTextRead 90
InsertChars 91
InsertFile 70
Jump 75
KillLine 65
Mark 71
MarkVert 71
ModifyOldCmd 84
MoveDown 66
MoveLeft 67
MoveRight 67
MoveUp 66

New 69
NewText 68
NewWin 81
Open 70
OpenClip 72
OpenDefs 86
PageForm 83
Paste 71
PasteVert 71
PlayRecScript 80
PredText 68
PredWin 81
Print 69
Quit 82
RecPlay 80
RecStart 80
RecStop 80
ReDo 91
RemFold 76
RemTab 74
RepeatCmd 84
Replace 74
ReplaceAll 74
Return 66
Save 70
SaveClip 72
SaveRecScript 80
ScreenDown 67
ScreenUp 67
SelTextN 69
SetBookMark 77
SetFoldsOff 76
SetFoldsOn 76
SetMode1, SetMode2, SetMode3 91
SetOverOff 78
SetOverOn 78
SetTab 78
SetWrap 78
SetWrapOff 78
SetWrapOn 78
ShowFold 76
SplitHorWin 81
SplitVerWin 81
SuccText 69
SuccWin 81
TextRead 90
ToggleKey 77
Top 68
UnDo 90
UndoLine 65
UpdateText 70

UpdateTexts 70
WrapPara 79
WrapText 79
Kommandozeile
Eingabe 47
erneut ausführen 47
letzte Eingabe 47
Kommandozeilen-Interpreter 47

L

Lesezeichen 33
LIST 64

M

Makros
abspielen 38
ARexx 39
aufzeichnen 37
speichern 38
MENU 56
Menü-Gruppe 56
MenuItem 56
ModeQualList 58

O

Open Defs 61
OPT_ 64
Overstrike 34

P

Parameter 64
Paste 20
Peppi 107
Pictogramm 18, 49
positionieren
Anfang 29
Ende 29

Q

Qualifier 57
QuotedString 55

R

Redo 62
RETURN 12
Right Margin 35
Rückgabewert 64

S

Schieberegler 14
Seitenformat 51
Seitennummer 51
SMALL_STRING 64
Sondertaste 58
Spaces 35
Spaltenblöcke 21
Speichern
  Texte 18
  Zusatzinformationen 18
Standard-Datei 93
Status-Anzeige 11
STRING 64
Struktur-Check 27
Suchen
  Groß-/Kleinbuchstaben 26
  vollständige Wörter 26
Suchmuster 26
Suchrichtung 26

T

TAB 12
Tab Width 35
Tabs 35
Tastaturbelegung 56
Text
  Drucken 17
  Erzeugen 15
  Freigeben 16
  Laden 19
  Löschen 17
  neu umbrechen 36
  Speichern 18
  Wahl 16
TITLE 56

U

Umbruchposition 36
Umwandeln 24
Undo 62
Undo-Depth 49
Undo-Funktion 49, 63
Undo-Tiefe 64
UndoLine 12
UndoMem 49
Unterbrechen 44

V

VALUE 64
Verlassen 45
vertauschen 27
Voreinstellungen 34

W

Word-Wrap 34, 36, 50
Workbench 9

Z

Zeichenfolge 27
Zeichensatz 49
Zeilen positionieren 29
Ziffernblock 13
Zusatzinformationen 50
Zwischenspeichern 21